CICLOS FORMATIVOS
GRADO SUPERIOR



SISTEMAS GESTORES DE BASES DE DATOS



MARÍA JESÚS RAMOS

ALICIA RAMOS

FERNANDO MONTERO





McGraw Hill

www.mhe.es/cf/informatica



Sistemas gestores de bases de datos

Mª Jesús Ramos Martín
Alicia Ramos Martín
Fernando Montero Rodríguez

Revisión técnica:
Eduardo Alcalde Lancharro



MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTAFÉ DE BOGOTÁ • SANTIAGO • SAÕ PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

Sistemas gestores de bases de datos – Grado Superior

Edificio Valrealty, 1.ª planta
Basauri, 17
28023 Aravaca (Madrid)

ISBN: 84-481-4879-7
Depósito legal:

Editor del proyecto: **Mariano García Díaz**
Editor: **Ángel Rodríguez Luengo**
Técnico editorial: **Alfredo Horas de Prado**
Diseño de cubierta: **Mercader de ideas**
Diseño interior: **_nuevacocina Comunicación**
Composición: **Estudio Grafimarque, S.L.**
Impresión:

Impreso en España – Printed in Spain



Presentación

Este libro está enfocado al módulo de *Sistemas Gestores de Bases de Datos,* del Ciclo Superior de Administración de Sistemas Informáticos. También es un libro de gran utilidad para todos los estudiantes y profesionales que quieran introducirse en el mundo de la administración de las bases de datos, y principalmente de la base de datos Oracle.

El libro está organizado en una serie de Unidades, en las que se establecen los objetivos a conseguir mediante unos contenidos eminentemente prácticos y procedimentales, sin olvidar los conocimientos de base teóricos que nuestros alumnos han de tener.

Va acompañado del CD del alumno, en el que se podrán encontrar: definiciones de tablas, casos prácticos resueltos, software de utilidad para hacer modelos de datos y el gestor de base de datos Oracle. El contenido del libro se distribuye en 13 unidades, las cuales se dividen en los siguientes bloques:

Primera parte: comprende las unidades 1 y 2 que exponen los conceptos fundamentales sobre bases de datos y modelos de datos, en especial el modelo de datos relacional.

Segunda parte: abarca las unidades 3 a 7 que exponen el *lenguaje SQL* que permite manipular y definir los datos almacenados en bases de datos relacionales, especificando lo que se debe hacer pero no cómo se debe hacer. Para aprender el lenguaje SQL, se ha utilizado el intérprete de sentencias SQL llamado *SQL*Plus*.

Tercera parte: formada por las unidades 8 a 11 que exponen la utilización del lenguaje de programación *PL/SQL* (Lenguaje Procedimental/SQL) que se utiliza para acceder a bases de datos Oracle. El lenguaje integra tanto estructuras procedimentales como el acceso a bases de datos; lo cual, permite desarrollar componentes que acceden a los datos almacenados en la base de datos.

Cuarta parte: incluye las unidades 12 y 13 que exponen los mecanismos para crear, mantener y administrar bases de datos Oracle.

Debemos añadir que la obra sigue un sistema de aprendizaje progresivo, la primera parte nos introduce al diseño y modelado de bases de datos, la segunda a la manipulación y consulta de los datos contenidos en ellas, en la tercera parte se estudia un lenguaje de programación de cuarta generación que nos permitirá desarrollar componentes para acceder a los datos de la base de datos y en la cuarta se crean y administran bases de datos utilizando el gestor Oracle.

Agradecimientos.

Queremos manifestar nuestro agradecimiento y gratitud a nuestro editor Mariano García y al equipo de producción, por el esfuerzo realizado para que esta obra haya podido salir adelante. También queremos agradecer la colaboración realizada por nuestro revisor técnico, Eduardo Alcalde.

Finalmente, expresamos nuestro agradecimiento a todos los que nos han animado para realizar este proyecto. Deseamos que esta obra sea del agrado de todos aquellos que se dedican a la docencia en informática y contribuya a una mejor formación de nuestros alumnos.



Sumario

1 Sistemas gestores de bases de datos

2 El modelo de datos relacional

3 Introducción a SQL

4 Funciones

5 Cláusulas avanzadas de selección

6 Manipulación de datos: INSERT, UPDATE y DELETE



Sistemas gestores de bases de datos

1

En esta unidad aprenderás a:

1. Describir las funciones y ventajas de un sistema gestor de bases de datos.
2. Describir la arquitectura interna de un sistema de bases de datos.
3. Distinguir los esquemas físico, conceptual y externo de una base de datos.
4. Identificar los componentes de un sistema gestor de bases de datos.
5. Describir las características de los distintos modelos lógicos de datos.
6. Trabajar con los distintos modelos de bases de datos.





1.1 Introducción

Definimos un **Sistema Gestor de Bases de Datos** o **SGBD**, también llamado DBMS (*Data Base Management System*) como una colección de datos relacionados entre sí, estructurados y organizados, y un conjunto de programas que acceden y gestionan esos datos. La colección de esos datos se denomina **Base de Datos** o **BD**, (*DB Data Base*).

Antes de aparecer los SGBD (década de los setenta), la información se trataba y se gestionaba utilizando los típicos sistemas de gestión de archivos que iban soportados sobre un sistema operativo. Éstos consistían en un conjunto de programas que definían y trabajaban sus propios datos. Los datos se almacenan en archivos y los programas manejan esos archivos para obtener la información. Si la estructura de los datos de los archivos cambia, todos los programas que los manejan se deben modificar; por ejemplo, un programa trabaja con un archivo de datos de alumnos, con una estructura o registro ya definido; si se incorporan elementos o campos a la estructura del archivo, los programas que utilizan ese archivo se tienen que modificar para tratar esos nuevos elementos. En estos sistemas de gestión de archivos, la definición de los datos se encuentra codificada dentro de los programas de aplicación en lugar de almacenarse de forma independiente, y además el control del acceso y la manipulación de los datos viene impuesto por los programas de aplicación.

Esto supone un gran inconveniente a la hora de tratar grandes volúmenes de información. Surge así la idea de separar los datos contenidos en los archivos de los programas que los manipulan, es decir, que se pueda modificar la estructura de los datos de los archivos sin que por ello se tengan que modificar los programas con los que trabajan. Se trata de estructurar y organizar los datos de forma que se pueda acceder a ellos con independencia de los programas que los gestionan.

Inconvenientes de un sistema de gestión de archivos:

- **Redundancia e inconsistencia de los datos,** se produce porque los archivos son creados por distintos programas y van cambiando a lo largo del tiempo, es decir, pueden tener distintos formatos y los datos pueden estar duplicados en varios sitios. Por ejemplo, el teléfono de un alumno puede aparecer en más de un archivo. La redundancia aumenta los costes de almacenamiento y acceso, y trae consigo la inconsistencia de los datos: las copias de los mismos datos no coinciden por aparecer en varios archivos.

- **Dependencia de los datos física-lógica,** o lo que es lo mismo, la estructura física de los datos (definición de archivos y registros) se encuentra codificada en los programas de aplicación. Cualquier cambio en esa estructura implica al programador identificar, modificar y probar todos los programas que manipulan esos archivos.

- **Dificultad para tener acceso a los datos,** proliferación de programas, es decir, cada vez que se necesite una consulta que no fue prevista en el inicio implica la necesidad de codificar el programa de aplicación necesario. Lo que se trata de probar es que los entornos convencionales de procesamiento de archivos no permiten recuperar los datos necesarios de una forma conveniente y eficiente.

- **Separación y aislamiento de los datos,** es decir, al estar repartidos en varios archivos, y tener diferentes formatos, es difícil escribir nuevos programas que aseguren la manipulación de los datos correctos. Antes se deberían sincronizar todos los archivos para que los datos coincidiesen.

- **Dificultad para el acceso concurrente,** pues en un sistema de gestión de archivos es complicado que los usuarios actualicen los datos simultáneamente. Las actualizaciones concurrentes pueden dar por resultado datos inconsistentes, ya que se puede acceder a los datos por medio de diversos programas de aplicación.

- **Dependencia de la estructura del archivo con el lenguaje de programación,** pues la estructura se define dentro de los programas. Esto implica que los formatos de los archivos sean incompatibles. La incompatibilidad entre archivos generados por distintos lenguajes hace que los datos sean difíciles de procesar.

- **Problemas en la seguridad de los datos.** Resulta difícil implantar restricciones de seguridad pues las aplicaciones se van añadiendo al sistema según se van necesitando.

- **Problemas de integridad de datos,** es decir, los valores almacenados en los archivos deben cumplir con restricciones de consistencia. Por ejemplo, no se puede insertar una nota de un alumno en una asignatura si previamente esa asignatura no está creada. Otro ejemplo, las unidades en almacén de un producto determinado no deben ser inferiores a una cantidad. Esto implica añadir gran número de líneas de código en los programas. El problema se complica cuando existen restricciones que implican varios datos en distintos archivos.

Todos estos inconvenientes hacen posible el fomento y desarrollo de SGBD. El objetivo primordial de un gestor es proporcionar eficiencia y seguridad a la hora de extraer o almacenar información en las BD. Los sistemas gestores de BBDD están diseñados para gestionar grandes bloques de información, que implica tanto la definición de estructuras para el almacenamiento como de mecanismos para la gestión de la información.

Una BD es un gran almacén de datos que se define una sola vez; los datos pueden ser accedidos de forma simultánea por varios usuarios; están relacionados y existe un número mínimo de duplicidad; además en las BBDD se almacenarán las descripciones de esos datos, lo que se llama *metadatos* en el diccionario de datos, que se verá más adelante.

El SGBD es una aplicación que permite a los usuarios definir, crear y mantener la BD y proporciona un acceso controlado a la misma. Debe prestar los siguientes servicios:

- **Creación y definición de la BD:** especificación de la estructura, el tipo de los datos, las restricciones y relaciones entre ellos mediante lenguajes de definición de datos. Toda esta información se almacena en el diccionario de datos, el SGBD proporcionará mecanismos para la gestión del diccionario de datos.

- **Manipulación de los datos** realizando consultas, inserciones y actualizaciones de los mismos utilizando lenguajes de manipulación de datos.

- **Acceso controlado a los datos de la BD** mediante mecanismos de seguridad de acceso a los usuarios.





- **Mantener la integridad y consistencia** de los datos utilizando mecanismos para evitar que los datos sean perjudicados por cambios no autorizados.

- **Acceso compartido a la BD,** controlando la interacción entre usuarios concurrentes.

- **Mecanismos de respaldo y recuperación** para restablecer la información en caso de fallos en el sistema.

1.2 Arquitectura de los sistemas de bases de datos

En 1975, el comité ANSI-SPARC (*American National Standard Institute - Standards Planning and Requirements Committee*) propuso una arquitectura de tres niveles para los SGBD cuyo objetivo principal era el de separar los programas de aplicación de la BD física. En esta arquitectura el esquema de una BD se define en tres niveles de abstracción distintos:

- **Nivel interno o físico:** el más cercano al almacenamiento físico, es decir, tal y como están almacenados en el ordenador. Describe la estructura física de la BD mediante un esquema interno. Este esquema se especifica con un modelo físico y describe los detalles de cómo se almacenan físicamente los datos: los archivos que contienen la información, su organización, los métodos de acceso a los registros, los tipos de registros, la longitud, los campos que los componen, etcétera.

- **Nivel externo o de visión:** es el más cercano a los usuarios, es decir, es donde se describen varios esquemas externos o vistas de usuarios. Cada esquema describe la parte de la BD que interesa a un grupo de usuarios en este nivel se representa la visión individual de un usuario o de un grupo de usuarios.

- **Nivel conceptual:** describe la estructura de toda la BD para un grupo de usuarios mediante un esquema conceptual. Este esquema describe las entidades, atributos, relaciones, operaciones de los usuarios y restricciones, ocultando los detalles de las estructuras físicas de almacenamiento. Representa la información contenida en la BD. En la Figura 1.1 se representan los niveles de abstracción de la arquitectura ANSI.

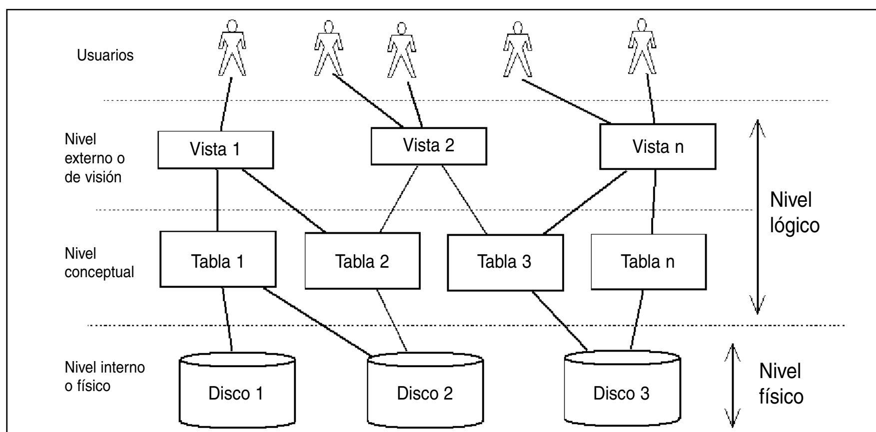


Figura 1.1. *Niveles de abstracción de la arquitectura ANSI.*

Esta arquitectura describe los datos a tres niveles de abstracción. En realidad los únicos datos que existen están a nivel físico almacenados en discos u otros dispositivos. Los SGBD basados en esta arquitectura permiten que cada grupo de usuarios haga referencia a su propio esquema externo. El SGBD debe de transformar cualquier petición de usuario (esquema externo) a una petición expresada en términos de esquema conceptual, para finalmente ser una petición expresada en el esquema interno que se procesará sobre la BD almacenada. El proceso de transformar peticiones y resultados de un nivel a otro se denomina correspondencia o transformación, el SGBD es capaz de interpretar una solicitud de datos y realiza los siguientes pasos:

- El usuario solicita unos datos y crea una consulta.
- El SGBD verifica y acepta el esquema externo para ese usuario.
- Transforma la solicitud al esquema conceptual.
- Verifica y acepta el esquema conceptual.
- Transforma la solicitud al esquema físico o interno.
- Selecciona la o las tablas implicadas en la consulta y ejecuta la consulta.
- Transforma del esquema interno al conceptual, y del conceptual al externo.
- Finalmente, el usuario ve los datos solicitados.

Para una BD específica sólo hay un esquema interno y uno conceptual, pero puede haber varios esquemas externos definidos para uno o para varios usuarios.

Con la arquitectura a tres niveles se introduce el concepto de independencia de datos, se definen dos tipos de independencia:

- **Independencia lógica:** la capacidad de modificar el esquema conceptual sin tener que alterar los esquemas externos ni los programas de aplicación. Se podrá modificar el esquema conceptual para ampliar la BD o para reducirla, por ejemplo, si se elimina una entidad, los esquemas externos que no se refieran a ella no se verán afectados.

- **Independencia física:** la capacidad de modificar el esquema interno sin tener que alterar ni el esquema conceptual, ni los externos. Por ejemplo, se pueden reorganizar los archivos físicos con el fin de mejorar el rendimiento de las operaciones de consulta o de actualización, o se pueden añadir nuevos archivos de datos porque los que había se han llenado. La independencia física es más fácil de conseguir que la lógica, pues se refiere a la separación entre las aplicaciones y las estructuras físicas de almacenamiento.

En los SGBD basados en arquitecturas de varios niveles se hace necesario ampliar el catálogo o el diccionario de datos para incluir la información sobre cómo establecer las correspondencias entre las peticiones de los usuarios y los datos, entre los diversos niveles. El SGBD utiliza una serie de procedimientos adicionales para realizar estas correspondencias haciendo referencia a la información de correspondencia que se encuentra




en el diccionario. La independencia de los datos se consigue porque al modificarse el esquema en algún nivel, el esquema del nivel inmediato superior permanece sin cambios. Sólo se modifica la correspondencia entre los dos niveles. No es preciso modificar los programas de aplicación que hacen referencia al esquema del nivel superior.

Sin embargo, los dos niveles de correspondencia implican un gasto de recursos durante la ejecución de una consulta o de un programa, lo que reduce la eficiencia del SGBD. Por esta razón pocos SGBD han implementado la arquitectura completa.

1.3 Componentes de los SGBD

Los SGBD son paquetes de software muy complejos que deben proporcionar una serie de servicios que van a permitir almacenar y explotar los datos de forma eficiente. Los componentes principales son los siguientes:

A. Lenguajes de los SGBD

Todos los SGBD ofrecen lenguajes e interfaces apropiadas para cada tipo de usuario: administradores, diseñadores, programadores de aplicaciones y usuarios finales.
Los lenguajes van a permitir al administrador de la BD especificar los datos que componen la BD, su estructura, las relaciones que existen entre ellos, las reglas de integridad, los controles de acceso, las características de tipo físico y las vistas externas de los usuarios. Los lenguajes del SGBD se clasifican en:

- **Lenguaje de definición de datos (LDD o DDL):** se utiliza para especificar el esquema de la BD, las vistas de los usuarios y las estructuras de almacenamiento. Es el que define el esquema conceptual y el esquema interno. Lo utilizan los diseñadores y los administradores de la BD.

- **Lenguaje de manipulación de datos (LMD o DML):** se utilizan para leer y actualizar los datos de la BD. Es el utilizado por los usuarios para realizar consultas, inserciones, eliminaciones y modificaciones. Los hay *procedurales*, en los que el usuario será normalmente un programador y especifica las operaciones de acceso a los datos llamando a los procedimientos necesarios. Estos lenguajes acceden a un registro y lo procesan. Las sentencias de un LMD procedural están embebidas en un lenguaje de alto nivel llamado *anfitrión*. Las BD jerárquicas y en red utilizan estos LMD procedurales.

  No procedurales son los lenguajes declarativos. En muchos SGBD se pueden introducir interactivamente instrucciones del LMD desde un terminal, también pueden ir embebidas en un lenguaje de programación de alto nivel. Estos lenguajes permiten especificar los datos a obtener en una consulta, o los datos a modificar, mediante sentencias sencillas. Las BD relacionales utilizan lenguajes no procedurales como SQL (*Structured Quero Language*) o QBE (*Query By Example*).

- La mayoría de los SGBD comerciales incluyen **lenguajes de cuarta generación (4GL)** que permiten al usuario desarrollar aplicaciones de forma fácil y rápida, también se les llama *herramientas de desarrollo*. Ejemplos de esto son las herramientas del SGBD

ORACLE: SQL Forms para la generación de formularios de pantalla y para interactuar con los datos; SQL Reports para generar informes de los datos contenidos en la BD; PL/SQL lenguaje para crear procedimientos que interractuen con los datos de la BD.

B. El diccionario de datos

El **diccionario de datos** es el lugar donde se deposita información acerca de todos los datos que forman la BD. Es una guía en la que se describe la BD y los objetos que la forman.

El diccionario contiene las características lógicas de los sitios donde se almacenan los datos del sistema, incluyendo nombre, descripción, alias, contenido y organización. Identifica los procesos donde se emplean los datos y los sitios donde se necesita el acceso inmediato a la información.

En una BD relacional, el diccionario de datos proporciona información acerca de:

- La estructura lógica y física de la BD.
- Las definiciones de todos los objetos de la BD: tablas, vistas, índices, disparadores, procedimientos, funciones, etcétera.
- El espacio asignado y utilizado por los objetos.
- Los valores por defecto de las columnas de las tablas.
- Información acerca de las restricciones de integridad.
- Los privilegios y roles otorgados a los usuarios.
- Auditoría de información, como los accesos a los objetos.

Un diccionario de datos debe cumplir las siguientes características:

- Debe soportar las descripciones de los modelos conceptual, lógico, interno y externo de la BD.
- Debe estar integrado dentro del SGBD.
- Debe apoyar la transferencia eficiente de información al SGDB. La conexión entre los modelos interno y externo debe ser realizada en tiempo de ejecución.
- Debe comenzar con la reorganización de versiones de producción de la BD. Además debe reflejar los cambios en la descripción de la BD. Cualquier cambio a la descripción de programas ha de ser reflejado automáticamente en la librería de descripción de programas con la ayuda del diccionario de datos.
- Debe estar almacenado en un medio de almacenamiento con acceso directo para la fácil recuperación de información.





C. Seguridad e integridad de datos

Un SGBD proporciona los siguientes mecanismos para garantizar la seguridad e integridad de los datos:

- Debe garantizar la protección de los datos contra accesos no autorizados, tanto intencionados como accidentales. Debe controlar que sólo los usuarios autorizados accedan a la BD.

- Los SGBD ofrecen mecanismos para implantar restricciones de integridad en la BD. Estas restricciones van a proteger la BD contra daños accidentales. Los valores de los datos que se almacenan deben satisfacer ciertos tipos de restricciones de consistencia y reglas de integridad, que especificará el administrador de la BD. El SGBD puede determinar si se produce una violación de la restricción.

- Proporciona herramientas y mecanismos para la planificación y realización de copias de seguridad y restauración.

- Debe ser capaz de recuperar la BD llevándola a un estado consistente en caso de ocurrir algún suceso que la dañe.

- Debe asegurar el acceso concurrente y ofrecer mecanismos para conservar la consistencia de los datos en el caso de que varios usuarios actualicen la BD de forma concurrente.

D. El administrador de la BD

En los sistemas de gestión de BBDD actuales existen diferentes categorías de usuarios. Estas categorías se caracterizan porque cada una de ellas tiene una serie de privilegios o permisos sobre los objetos que forman la BD.

En los sistemas Oracle las categorías más importantes son:

- **Los usuarios de la categoría DBA** (*Database Administrator*), cuya función es precisamente administrar la base y que tienen, el nivel más alto de privilegios.

- **Los usuarios de la categoría RESOURCE,** que pueden crear sus propios objetos y tienen acceso a los objetos para los que se les ha concedido permiso.

- **Los usuarios del tipo CONNECT,** que solamente pueden utilizar aquellos objetos para los que se les ha concedido permiso de acceso.

El DBA tiene una gran responsabilidad ya que posee el máximo nivel de privilegios. Será el encargado de crear los usuarios que se conectarán a la BD. En la administración de una BD siempre hay que procurar que haya el menor número de administradores, a ser posible una sola persona.

El objetivo principal de un DBA es garantizar que la BD cumple los fines previstos por la organización, lo que incluye una serie de tareas como:

- Instalar SGBD en el sistema informático.
- Crear las BBDD que se vayan a gestionar.
- Crear y mantener el esquema de la BD.
- Crear y mantener las cuentas de usuario de la BD.
- Arrancar y parar SGBD, y cargar las BBDD con las que se ha de trabajar.
- Colaborar con el administrador del S.O. en las tareas de ubicación, dimensionado y control de los archivos y espacios de disco ocupados por el SGBD.
- Colaborar en las tareas de formación de usuarios.
- Establecer estándares de uso, políticas de acceso y protocolos de trabajo diario para los usuarios de la BD.
- Suministrar la información necesaria sobre la BD a los equipos de análisis y programación de aplicaciones.
- Efectuar tareas de explotación como:
  - Vigilar el trabajo diario colaborando en la información y resolución de las dudas de los usuarios de la BD.
  - Controlar en tiempo real los accesos, tasas de uso, cargas en los servidores, anomalías, etcétera.
  - Llegado el caso, reorganizar la BD.
  - Efectuar las copias de seguridad periódicas de la BD.
  - Restaurar la BD después de un incidente material a partir de las copias de seguridad.
  - Estudiar las auditorías del sistema para detectar anomalías, intentos de violación de la seguridad, etcétera.
  - Ajustar y optimizar la BD mediante el ajuste de sus parámetros, y con ayuda de las herramientas de monitorización y de las estadísticas del sistema.

En su gestión diaria, el DBA suele utilizar una serie de herramientas de administración de la BD.

Con el paso del tiempo, estas herramientas han adquirido sofisticadas prestaciones y facilitan en gran medida la realización de trabajos que, hasta no hace demasiado, requerían de arduos esfuerzos por parte de los administradores.





1.4 Modelos de datos

Uno de los objetivos más importantes de un SGBD es proporcionar a los usuarios una visión abstracta de los datos, es decir, el usuario va a utilizar esos datos pero no tendrá idea de cómo están almacenados físicamente.

Los modelos de datos son el instrumento principal para ofrecer esa abstracción. Son utilizados para la representación y el tratamiento de los problemas. Forman el problema a tres niveles de abstracción, relacionados con la arquitectura ANSI-SPARC de tres niveles para los SGBD:

- **Nivel físico:** el nivel más bajo de abstracción; describe cómo se almacenan realmente los datos.
- **Nivel lógico o conceptual:** describe los datos que se almacenan en la BD y sus relaciones, es decir, los objetos del mundo real, sus atributos y sus propiedades, y las relaciones entre ellos.
- **Nivel externo o de vistas:** describe la parte de la BD a la que los usuarios pueden acceder.

Para hacernos una idea de los tres niveles de abstracción, nos imaginamos un archivo de artículos con el siguiente registro:

```
struct ARTICULOS
 {  int Cod;
    char Deno[15];
    int cant_almacen;
    int cant_minima ;
    int uni_vendidas;
    float PVP;
    char reponer;
    struct VENTAS Tventas[12];
 };
```

El **nivel físico** es el conjunto de bytes que se encuentran almacenados en el archivo en un dispositivo magnético, que puede ser un disco, una pista a un sector determinado.

El **nivel lógico** comprende la descripción y la relación con otros registros que se hace del registro dentro de un programa, en un lenguaje de programación.

El último nivel de abstracción, el **externo**, es la visión de estos datos que tiene un usuario cuando ejecuta aplicaciones que operan con ellos, el usuario no sabe el detalle de los datos, unas veces operará con unos y otras con otros, dependiendo de la aplicación.

Si trasladamos el ejemplo a una BD relacional específica habrá, como en el caso anterior, un único nivel interno y un único nivel lógico o conceptual, pero puede haber varios niveles externos, cada uno definido para uno o para varios usuarios. Podría ser el siguiente:

Curso	Nombre	Nombre de asignatura	Nota
1	Ana	Programación en lenguajes estructurados	6
1	Ana	Sistemas informáticos multiusuario y en red	8
2	Rosa	Desa. de aplic. en entornos de 4.ª Generación y H. Case	5
2	Juan	Desa. de aplic. en entornos de 4.ª Generación y H. Case	7
1	Alicia	Programación en lenguajes estructurados	5
1	Alicia	Sistemas informáticos multiusuario y en red	4

Tabla 1.1. *Vista de la BD para un usuario.*

- **Nivel externo:** Visión parcial de las tablas de la BD según el usuario. Por ejemplo, la vista que se muestra en la Tabla 1.1 obtiene el listado de notas de alumnos con los siguientes datos: Curso, Nombre, Nombre de asignatura y Nota.

- **Nivel lógico y conceptual:** Definición de todas las tablas, columnas, restricciones, claves y relaciones. En este ejemplo, disponemos de tres tablas que están relacionadas:

  – *Tabla ALUMNOS*. Columnas: NMatrícula, Nombre, Curso, Dirección, Población. Clave: NMatrícula. Además tiene una relación con NOTAS, pues un alumno puede tener notas en varias asignaturas.

  – *Tabla ASIGNATURAS*. Columnas: Código, Nombre de asignatura. Clave: Código. Está relacionada con NOTAS, pues para una asignatura hay varias notas, tantas como alumnos la cursen.

  – *Tabla NOTAS*. Columnas: NMatrícula, Código, Nota. Está relacionada con ALUMNOS y ASIGNATURAS, pues un alumno tiene notas en varias asignaturas, y de una asignatura existen varias notas, tantas como alumnos.

Podemos representar las relaciones de las tablas en el nivel lógico como se muestra en la Figura 1.2:



Figura 1.2. *Representación de las relaciones entre tablas en el nivel lógico.*

- **Nivel interno**: En una BD las tablas se almacenan en archivos de datos de la BD. Si hay claves, se crean índices para acceder a los datos, todo esto contenido en el disco duro, en una pista y en un sector, que sólo el SGBD conoce. Ante una petición, sabe a qué pista, a qué sector, a qué archivo de datos y a qué índices acceder.





Para la representación de estos niveles se utilizan los *modelos de datos*. Se definen como el conjunto de conceptos o herramientas conceptuales que sirven para describir la estructura de una BD: los datos, las relaciones y las restricciones que se deben cumplir sobre los datos. Se denomina **esquema de la BD** a la descripción de una BD mediante un modelo de datos. Este esquema se especifica durante el diseño de la misma.

Podemos dividir los modelos en tres grupos: *modelos lógicos basados en objetos*, *modelos lógicos basados en registros* y *modelos físicos de datos*. Cada SGBD soporta un modelo lógico.

Modelos lógicos basados en objetos

Los modelos lógicos basados en objetos se usan para describir datos en el nivel conceptual y el externo. Se caracterizan porque proporcionan capacidad de estructuración bastante flexible y permiten especificar restricciones de datos. Los modelos más conocidos son el modelo *entidad-relación* y el *orientado a objetos.*

Actualmente, el más utilizado es el modelo entidad-relación, aunque el modelo orientado a objetos incluye muchos conceptos del anterior, y poco a poco está ganando mercado. La mayoría de las BBDD relacionales añaden extensiones para poder ser relacionales-orientadas a objetos.

Modelos lógicos basados en registros

Los modelos lógicos basados en registros se utilizan para describir los datos en los modelos conceptual y físico. A diferencia de los modelos lógicos basados en objetos, se usan para especificar la estructura lógica global de la BD y para proporcionar una descripción a nivel más alto de la implementación.

Los modelos basados en registros se llaman así porque la BD está estructurada en registros de formato fijo de varios tipos. Cada tipo de registro define un número fijo de campos, o atributos, y cada campo normalmente es de longitud fija. La estructura más rica de estas BBDD a menudo lleva a registros de longitud variable en el nivel físico.

Los modelos basados en registros no incluyen un mecanismo para la representación directa de código de la BD, en cambio, hay lenguajes separados que se asocian con el modelo para expresar consultas y actualizaciones. Los tres modelos de datos más aceptados son los modelos *relacional*, *de red* y *jerárquico*. El modelo relacional ha ganado aceptación por encima de los otros; representa los datos y las relaciones entre los datos mediante una colección de tablas, cuyas columnas tienen nombres únicos, las filas (*tuplas*) representan a los registros y las columnas representan las características (*atributos*) de cada registro. Este modelo se estudiará en la siguiente Unidad.

Modelos físicos de datos

Los modelos físicos de datos se usan para describir cómo se almacenan los datos en el ordenador: formato de registros, estructuras de los archivos, métodos de acceso, etcétera. Hay muy pocos modelos físicos de datos en uso, siendo los más conocidos el modelo *unificador* y *de memoria de elementos*.

1.5 El modelo entidad-interrelación

El modelo de datos entidad-interrelación (E-R), también llamado entidad-relación, fue propuesto por Peter Chen en 1976 para la representación conceptual de los problemas del mundo real. En 1988, el ANSI lo seleccionó como modelo estándar para los sistemas de diccionarios de recursos de información. Es un modelo muy extendido y potente para la representación de los datos. Se simboliza haciendo uso de grafos y de tablas. Propone el uso de tablas bidimensionales para la representación de los datos y sus relaciones.

Conceptos básicos

Entidad. Es un objeto del mundo real, que tiene interés para la empresa. Por ejemplo, los ALUMNOS de un centro escolar o los CLIENTES de un banco. Se representa utilizando rectángulos.

Conjunto de entidades. Es un grupo de entidades del mismo tipo, por ejemplo, el conjunto de entidades cliente. Los conjuntos de entidades no necesitan ser disjuntos, se puede definir los conjuntos de entidades de empleados y clientes de un banco, pudiendo existir una persona en ambas o ninguna de las dos cosas.

Entidad fuerte. Es aquella que no depende de otra entidad para su existencia. Por ejemplo, la entidad ALUMNO es fuerte pues no depende de otra para existir, en cambio, la entidad NOTAS es una entidad débil pues necesita a la entidad ALUMNO para existir. Las entidades débiles se relacionan con la entidad fuerte con una relación uno a varios. Se representan con un rectángulo con un borde doble.

Atributos o campos. Son las unidades de información que describen propiedades de las entidades. Por ejemplo, la entidad ALUMNO posee los atributos: número de matrícula, nombre, dirección, población y teléfono. Los atributos toman valores, por ejemplo, el atributo población puede ser ALCALÁ, GUADALAJARA, etcétera. Se representan mediante una elipse con el nombre en su interior.

Dominio. Es el conjunto de valores permitido para cada atributo. Por ejemplo el dominio del atributo nombre puede ser el conjunto de cadenas de texto de una longitud determinada.

Identificador o superclave. Es el conjunto de atributos que identifican de forma única a cada entidad. Por ejemplo, la entidad EMPLEADO, con los atributos Número de la Seguridad Social, DNI, Nombre, Dirección, Fecha nacimiento y Tlf, podrían ser identificadores o superclaves los conjuntos Nombre, Dirección, Fecha nacimiento y Tlf, o también DNI, Nombre y Dirección, o también Num Seg Social, Nombre, Dirección y Tlf, o solos el DNI y el Número de la Seguridad Social.

Clave candidata. Es cada una de las superclaves formadas por el mínimo número de campos posibles. En el ejemplo anterior, son el DNI y el Número de la Seguridad Social.

Clave primaria o principal (*primary key*)**:** Es la clave candidata seleccionada por el diseñador de la BD. Una clave candidata no puede contener valores nulos, ha de ser sencilla de crear y no ha de variar con el tiempo. El atributo o los atributos que forman esta clave se representan subrayados.



Clave ajena o foránea (*foreign key*)**:** Es el atributo o conjunto de atributos de una entidad que forman la clave primaria en otra entidad. Las claves ajenas van a representar las relaciones entre tablas. Por ejemplo, si tenemos por un lado, las entidades ARTÍCULOS, con los atributos código de artículo (clave primaria), denominación, stock. Y, por otro lado, VENTAS, con los atributos Código de venta (clave primaria), fecha de venta, código de artículo, unidades vendidas, el código de artículo es clave ajena pues está como clave primaria en la entidad ARTÍCULOS.

A. Relaciones y conjuntos de relaciones

Definimos una **relación** como la asociación entre diferentes entidades. Tienen nombre de verbo, que la identifica de las otras relaciones y se representa mediante un rombo. Normalmente las relaciones no tienen atributos. Cuando surge una relación con atributos significa que debajo hay una entidad que aún no se ha definido. A esa entidad se la llama *entidad asociada*. Esta entidad dará origen a una tabla que contendrá esos atributos. Esto se hace en el *modelo relacional* a la hora de representar los datos. Lo veremos más adelante.

Un **conjunto de relaciones** es un conjunto de relaciones del mismo tipo, por ejemplo entre ARTÍCULOS y VENTAS todas las asociaciones existentes entre los artículos y las ventas que tengan estos, forman un conjunto de relaciones.

La mayoría de los conjuntos de relaciones en un sistema de BD son binarias (dos entidades) aunque puede haber conjuntos de relaciones que implican más de dos conjuntos de entidades, por ejemplo, una relación como la relación entre cliente, cuenta y sucursal. Siempre es posible sustituir un conjunto de relaciones no binario por varios conjuntos de relaciones binarias distintos. Así, conceptualmente, podemos restringir el modelo E-R para incluir sólo conjuntos binarios de relaciones, aunque no siempre es posible.

La función que desempeña una entidad en una relación se llama *papel*, y normalmente es implícito y no se suele especificar. Sin embargo, son útiles cuando el significado de una relación necesita ser clarificado.

Una relación también puede tener atributos descriptivos, por ejemplo, la FECHA_OPERACIÓN en el conjunto de relaciones CLIENTE_CUENTA, que especifica la última fecha en la que el cliente tuvo acceso a su cuenta (ver Figura 1.3).

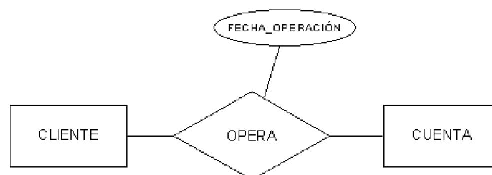


Figura 1.3. *Relación con atributos descriptivos.*

Diagramas de estructuras de datos en el modelo E-R

Los diagramas Entidad-Relación representan la estructura lógica de una BD de manera gráfica. Los símbolos utilizados son los siguientes:

- Rectángulos para representar a las entidades.

– Elipses para los atributos. El atributo que forma parte de la clave primaria va subrayado.

– Rombos para representar las relaciones.

– Las líneas, que unen atributos a entidades y a relaciones, y entidades a relaciones. Si la flecha tiene punta, en ese sentido está el uno, y si no la tiene, en ese sitio está el muchos. La orientación señala cardinalidad.

– Si la relación tiene atributos asociados, se le unen a la relación.

– Cada componente se etiqueta con el nombre de lo que representa.

En la Figura 1.4 se muestra un diagrama E-R correspondiente a PROVEEDORES-ARTÍCULOS. Un PROVEEDOR SUMINISTRA muchos ARTÍCULOS.

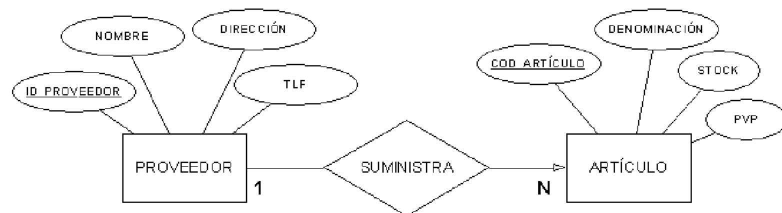


Figura 1.4. *Diagrama E-R, un proveedor suministra muchos artículos.*

Grado y cardinalidad de las relaciones

Se define **grado de una relación** como el número de conjuntos de entidades que participan en el conjunto de relaciones, o lo que es lo mismo, el número de entidades que participan en una relación. Las relaciones en las que participan dos entidades son *binarias* o *de grado dos*. Si participan tres serán *ternarias* o *de grado 3*. Los conjuntos de relaciones pueden tener cualquier grado, lo ideal es tener relaciones binarias.

Las relaciones en las que sólo participa una entidad se llaman *anillo* o *de grado uno*; relaciona una entidad consigo misma, se las llama *relaciones reflexivas*. Por ejemplo, la entidad EMPLEADO puede tener una relación JEFE DE consigo misma: un empleado es JEFE DE muchos empleados y, a la vez, el jefe es un empleado.

Otro ejemplo puede ser la relación DELEGADO DE los alumnos de un curso: el delegado es alumno también del curso. Ver Figura 1.5.

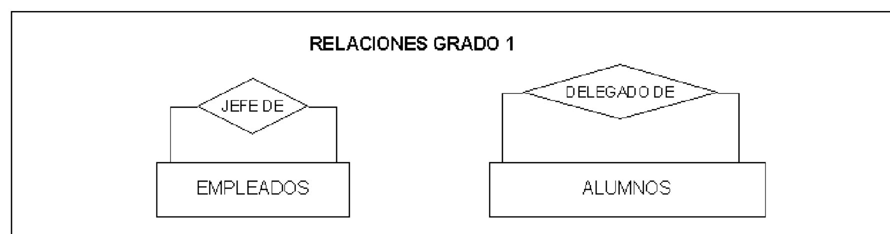


Figura 1.5. *Relaciones de grado 1.*



En la Figura 1.6 se muestra una relación de grado dos, que representa un proveedor que suministra artículos, y otra de grado tres, que representa un cliente de un banco que tiene varias cuentas, y cada una en una sucursal:

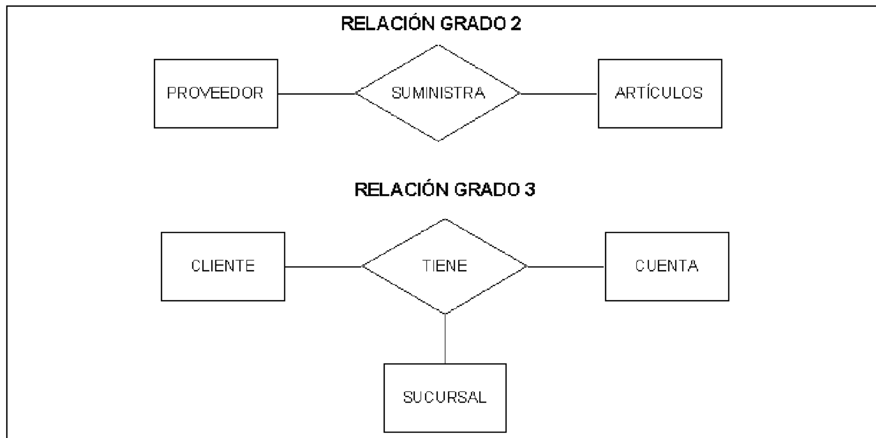


Figura 1.6. *Relaciones de grados 2 y 3.*

En el modelo E-R se representan ciertas restricciones a las que deben ajustarse los datos contenidos en una BD. Éstas son las restricciones de las cardinalidades de asignación, que expresan el número de entidades a las que puede asociarse otra entidad mediante un conjunto de relación.

Las cardinalidades de asignación se describen para conjuntos binarios de relaciones. Son las siguientes:

- **1:1, uno a uno.** A cada elemento de la primera entidad le corresponde sólo uno de la segunda entidad, y a la inversa. Por ejemplo, un cliente de un hotel ocupa una habitación, o un curso de alumnos pertenece a un aula, y a esa aula sólo asiste ese grupo de alumnos. Ver Figura 1.7:

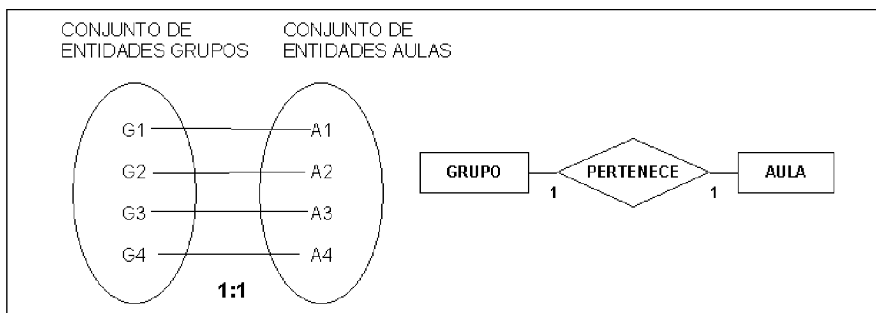


Figura 1.7. *Representación de relaciones uno a uno.*

- **1:N, uno a muchos.** A cada elemento de la primera entidad le corresponde uno o más elementos de la segunda entidad, y a cada elemento de la segunda entidad le corresponde uno sólo de la primera entidad. Por ejemplo, un proveedor suministra muchos artículos (ver Figura 1.8).

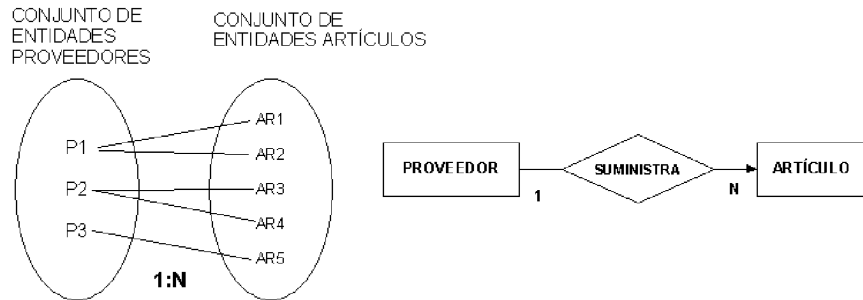


Figura 1.8. *Representación de relaciones uno a muchos.*

- **N:1, muchos a uno.** Es el mismo caso que el anterior pero al revés; a cada elemento de la primera entidad le corresponde un elemento de la segunda, y a cada elemento de la segunda entidad, le corresponden varios de la primera.

- **M:N, muchos a muchos.** A cada elemento de la primera entidad le corresponde uno o más elementos de la segunda entidad, y a cada elemento de la segunda entidad le corresponde uno o más elementos de la primera entidad. Por ejemplo, un vendedor vende muchos artículos, y un artículo es vendido por muchos vendedores (ver Figura 1.9).

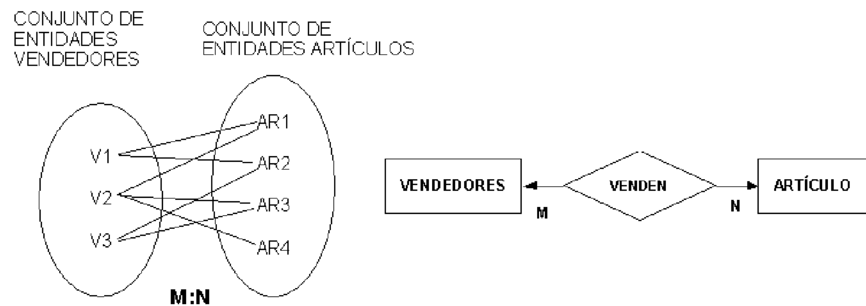


Figura 1.9. *Representación de relaciones muchos a muchos.*

La cardinalidad de una entidad sirve para conocer su grado de participación en la relación, es decir, el número de correspondencias en las que cada elemento de la entidad interviene. Mide la obligatoriedad de correspondencia entre dos entidades.

La representamos entre paréntesis indicando los valores máximo y mínimo: (máximo, mínimo). Los valores para la cardinalidad son: (0,1), (1,1), (0,N), (1,N) y (M,N). El valor 0 se pone cuando la participación de la entidad es opcional.

En la Figura 1.10, que se muestra a continuación, se representa el diagrama E-R en el que contamos con las siguientes entidades:

- EMPLEADO está formada por los atributos Nº. Emple, Apellido, Salario y Comisión, siendo el atributo Nº. Emple la clave principal (representado por el subrayado).

- DEPARTAMENTO está formada por los atributos Nº. Depart, Nombre y Localidad, siendo el atributo Nº. Depart la clave principal.



- Se han definido dos relaciones:

  – La relación «PERTENECE» entre las entidades EMPLEADOS y DEPARTAMENTO, cuyo tipo de correspondencia es 1:N, es decir, a un departamento le pertenecen cero o más empleados (0,N). Un empleado pertenece a un departamento y sólo a uno (1,1).

  – La relación «JEFE», que asocia la entidad EMPLEADO consigo misma. Su tipo de correspondencia es 1:N, es decir, un empleado es jefe de cero o más empleados (0,N). Un empleado tiene un jefe y sólo uno (1,1). Ver Figura 1.10:

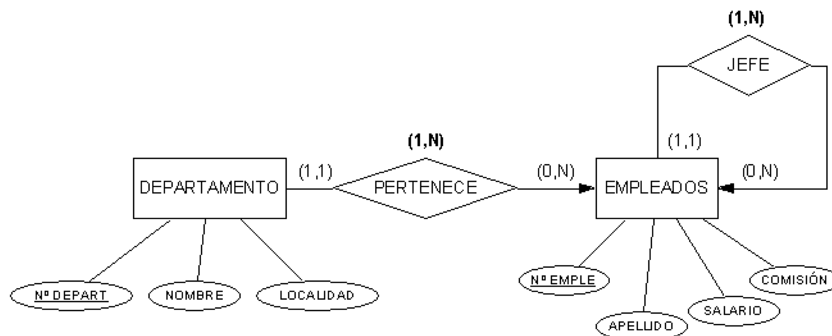


Figura 1.10. *Diagrama E-R de las relaciones entre departamentos y empleados.*

Caso práctico

1 **Vamos a realizar el diagrama de estructuras de datos en el modelo E-R. Supongamos que en un centro escolar se imparten muchos cursos. Cada curso está formado por un grupo de alumnos, de los cuales uno de ellos es el delegado del grupo. Los alumnos cursan asignaturas, y una asignatura puede o no ser cursada por los alumnos.**

Para su resolución, primero identificaremos las entidades, luego las relaciones y las cardinalidades y, por último, los atributos de las entidades y de las interrelaciones, si las hubiera.

1. Identificación de entidades: una entidad es un objeto del mundo real, algo que tiene interés para la empresa. Se hace un análisis del enunciado, de donde sacaremos los candidatos a entidades: CENTROS, CURSOS, ALUMNOS, ASIGNATURAS, DELEGADOS. Si analizamos esta última veremos que los delegados son alumnos, por lo tanto, los tenemos recogidos en ALUMNOS. Esta posible entidad la eliminaremos. También eliminaremos la posible entidad CENTROS pues se trata de un único centro, si se tratara de una gestión de centros tendría más sentido incluirla.

2. Identificar las relaciones: construimos una matriz de entidades en la que las filas y las columnas son los nombres de entidades y cada celda puede contener o no la relación, las relaciones aparecen explícitamente en el enunciado. En este ejemplo, las relaciones no tienen atributos. Del enunciado sacamos lo siguiente:

   – Un curso está formado por muchos alumnos. La relación entre estas dos entidades la llamamos PERTENECE, pues a un curso pertenecen muchos alumnos, relación 1:M. Consideramos que es obligatorio que existan alumnos en un curso. Para calcular los máximos y mínimos hacemos la pregunta: a un CURSO, ¿cuántos ALUMNOS pertenecen, como mínimo y como máximo? Y se ponen los valores en la entidad ALUMNOS, en este caso (1,M). Para el sentido contrario, hacemos lo mismo: un ALUMNO, ¿a cuántos CURSOS va a pertenecer? Como mínimo a 1, y como máximo a 1, en este caso pondremos (1,1) en la entidad CURSOS.

(Continúa)

(Continuación)

- De los alumnos que pertenecen a un grupo, uno de ellos es DELEGADO. Hay una relación de grado 1 entre la entidad ALUMNO que la podemos llamar ES DELEGADO. La relación es 1:M, un alumno es delegado de muchos alumnos. Para calcular los valores máximos y mínimos preguntamos: ¿un ALUMNO de cuántos alumnos ES DELEGADO? Como mínimo es 0, pues puede que no sea delegado, y como máximo es M, pues si es delegado lo será de muchos; pondremos en el extremo (0,M). Y en el otro extremo pondremos (1,1), pues obligatoriamente el delegado es un alumno.

- Entre ALUMNOS y ASIGNATURAS surge una relación N:M, pues un alumno cursa muchas asignaturas y una asignatura es cursada por muchos alumnos. La relación se llamará CURSA. Consideramos que puede haber asignaturas sin alumnos. Las cardinalidades serán (1:M) entre ALUMNO-ASIGNATURA, pues un alumno, como mínimo, cursa una asignatura, y, como máximo, muchas. La cardinalidad entre ASIGNATURA-ALUMNO será (0,N), pues una ASIGNATURA puede ser cursada por 0 alumnos o por muchos.

En la Tabla 1.2 se muestra la matriz de entidades y relaciones entre ellas:

	CURSOS	ALUMNOS	ASIGNATURAS
CURSOS	---------------	PERTENECE (1:M)	---------------
ALUMNOS	x	ES DELEGADO(1:M)	CURSA(N:M)
ASIGNATURAS	---------------	x	---------------

Tabla 1.2. *Matriz de entidades y relaciones entre ellas.*

Las celdas que aparecen con una x indican que las relaciones están ya identificadas. Las que aparecen con guiones indican que no existe relación. En la siguiente figura se muestra el diagrama de las relaciones y las cardinalidades.

3. Identificar los atributos, como el enunciado no explicita ningún tipo de característica de las entidades nos imaginamos los atributos, que pueden ser los siguientes:

CURSOS - COD_CURSO (clave primaria), DESCRIPCIÓN, NIVEL, TURNO y ETAPA

ALUMNOS - NUM-MATRÍCULA (clave primaria), NOMBRE, DIRECCIÓN, POBLACIÓN, TLF y NUM_HERMANOS

ASIGNATURAS - COD-ASIGNATURA (clave primaria), DENOMINACIÓN y TIPO

En la Figura 1.11 se representa el diagrama de estructuras del ejercicio:

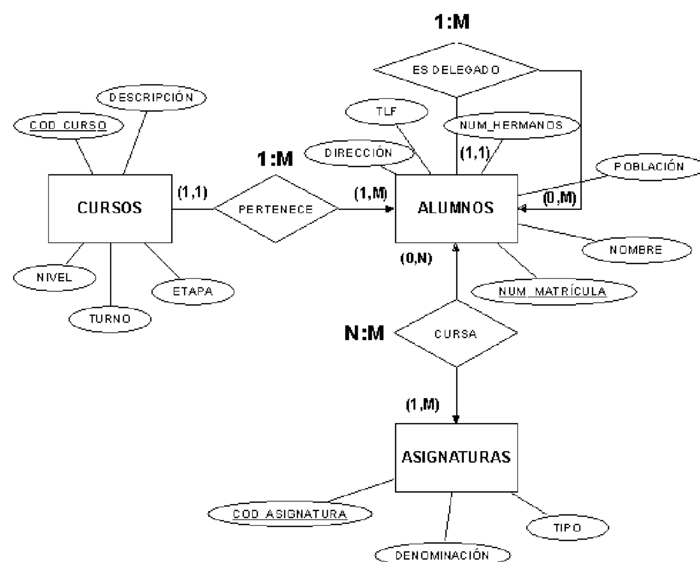


Figura 1.11. *Diagrama de estructuras en el modelo E-R.*





Actividades propuestas



1 Se desea realizar el diagrama de estructuras de datos en el modelo E-R, correspondiente al siguiente enunciado:

Supongamos el bibliobús que proporciona un servicio de préstamos de libros a los socios de un pueblo. Los libros están clasificados por temas. Un tema puede contener varios libros. Un libro es prestado a muchos socios, y un socio puede coger varios libros. En el préstamo de libros es importante saber la Fecha de préstamo y la Fecha de devolución. De los libros nos interesa saber el título, el autor y el número de ejemplares.

Generalización y jerarquías de generalización

Las generalizaciones proporcionan un mecanismo de abstracción que permite especializar una entidad (que se denominará *supertipo*) en subtipos, o lo que es lo mismo generalizar los subtipos en el supertipo.

Una generalización se identifica si encontramos una serie de atributos comunes a un conjunto de entidades, y unos atributos específicos que identificarán unas características.

Los atributos comunes describirán el supertipo y los particulares los subtipos. Una de las características más importantes de las jerarquías es la herencia, por la que los atributos de un supertipo son heredados por sus subtipos. Si el supertipo participa en una relación los subtipos también participarán.

Por ejemplo, en una empresa de construcción podremos identificar las siguientes entidades:

- EMPLEADO, con los atributos N_EMPLE (clave primaria,) NOMBRE, DIRECCIÓN, FECHA_NAC, SALARIO y PUESTO.

- ARQUITECTO, con los atributos de empleado más los atributos específicos: COMISIONES, y NUM_PROYECTOS.

- ADMINISTRATIVO, con los atributos de empleado más los atributos específicos: PULSACIONES y NIVEL.

- INGENIERO, con los atributos de empleado más los atributos específicos: ESPECIALIDAD y AÑOS_EXPERIENCIA.

En la Figura 1.12 se representa este ejemplo de generalización.

La generalización es total si no hay ocurrencias en el supertipo que no pertenezcan a ninguno de los subtipos, es decir, que los empleados o son arquitectos, o son administrativos, o son aparejadores, no pueden ser varias cosas a la vez. En este caso, la generalización sería también exclusiva. Si un empleado puede ser varias cosas a la vez la generalización es *solapada* o *superpuesta*.

La generalización es *parcial* si existen empleados que no son ni ingenieros, ni administrativos, ni arquitectos. También puede ser *exclusiva* o *solapada*. Las cardinalidades en estas relaciones son siempre (1,1) en el supertipo y (0,1) en los subtipos, para las exclusivas. (0,1) o (1,1) en los subtipos para las solapadas o superpuestas.

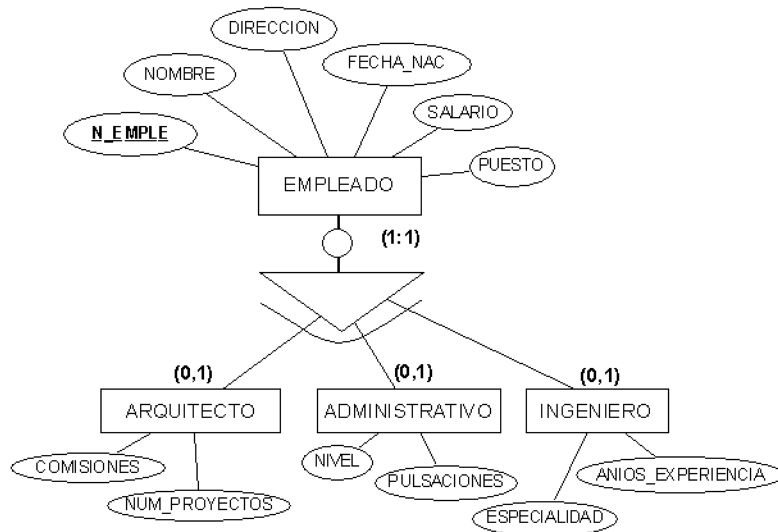


Figura 1.12. *Representación de una generalización.*

Así pues, habrá jerarquía solapada y parcial (que es la que no tiene ninguna restricción) solapada y total, exclusiva y parcial, y exclusiva y total. En la Figura 1.13 se muestran cómo se representan.

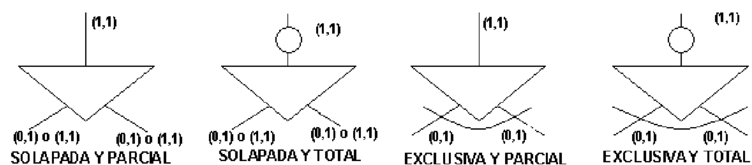


Figura 1.13. *Tipos y representación de jerarquías.*



Actividades propuestas

2 Representa las siguientes jerarquías e indica el tipo de generalización:

Un concesionario de coches vende coches nuevos y usados. Los atributos específicos de los nuevos son las unidades y el descuento; de los usados son los kilómetros y el año de fabricación.

Consideramos el conjunto de personas de una ciudad, distinguimos a los trabajadores, estudiantes y parados. De los trabajadores nos interesa el número de la Seguridad Social, la empresa de trabajo y el salario. De los estudiantes, el número de matrícula y el centro educativo, y de los parados la fecha del paro.

En un campo de fútbol los puestos de los futbolistas pueden ser: portero, defensa, medio y delantero.



Agregación

Una limitación del modelo E-R es que no es posible expresar relaciones entre relaciones. En estos casos se realiza una agregación, que es una abstracción a través de la cual las relaciones se tratan como entidades de nivel más alto. Por ejemplo, consideramos una relación entre EMPLEADOS y PROYECTOS, un empleado trabaja en varios proyectos durante unas horas determinadas y en ese trabajo utiliza unas herramientas determinadas. La representación del diagrama de estructuras se muestra en la Figura 1.14 siguiente:

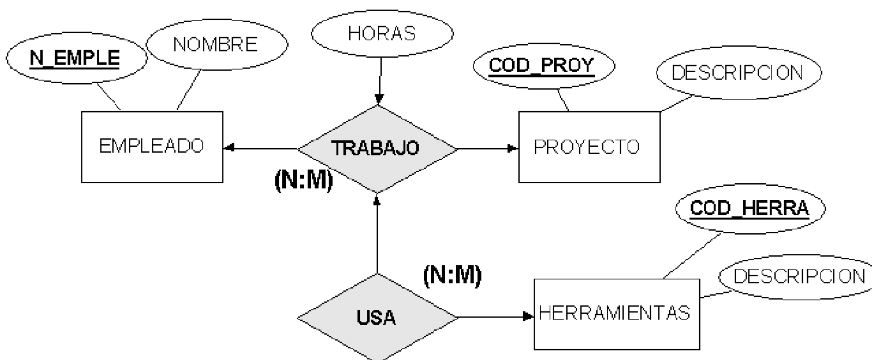


Figura 1.14. *Diagrama E-R de una relación entre otra relación.*

Si consideramos la agregación, tenemos que la relación TRABAJO con las entidades EMPLEADO y PROYECTO se pueden representar como un conjunto de entidades llamadas TRABAJO, que se relacionan con la entidad HERRAMIENTAS mediante la relación USA. Ver Figura 1.15:

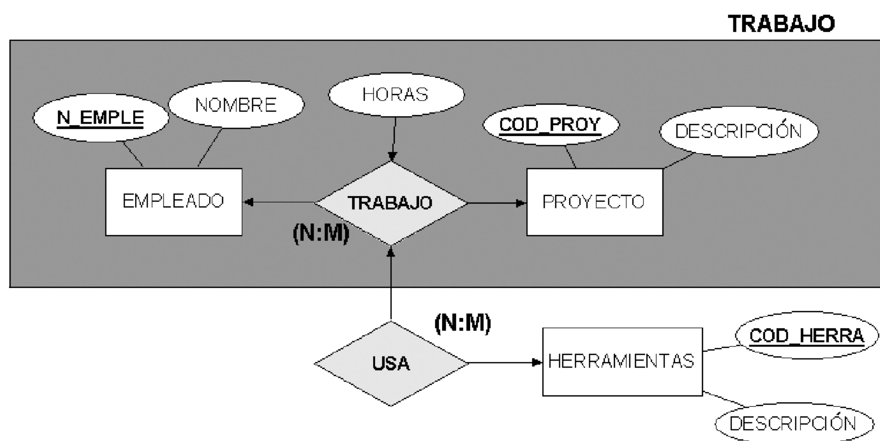


Figura 1.15. *Conjunto de entidades y relaciones para representar una relación entre una relación.*



Actividades propuestas

3 **Se desea realizar el diagrama de estructuras de datos en el modelo E-R correspondiente al siguiente enunciado:**

Una compañía de distribución de productos para el hogar dispone de proveedores que le suministran artículos. Un artículo sólo puede proveerlo un proveedor.

La empresa tiene tres tipos de empleados: oficinistas, transportistas y vendedores. Estos últimos venden los artículos. Un artículo es vendido por varios vendedores, y un vendedor puede vender varios artículos en distintas zonas de venta. De las ventas nos interesa saber la fecha de venta y las unidades vendidas.

1.6 Modelo de red

Este modelo utiliza estructuras de datos en red, también conocidas como *estructuras plex*. Las entidades se representan como registros o nodos, y las relaciones como enlaces o punteros. En una estructura red cualquier componente puede vincularse con cualquier otro. Es posible describirla en términos de padres e hijos, pero, a diferencia del modelo jerárquico, un hijo puede tener varios padres.

Las representaciones lógicas basadas en árboles o en estructuras plex, a menudo, limitan el cambio que el crecimiento de la BD exige, hasta tal punto que las representaciones lógicas de los datos pueden variar afectando a los programas de aplicación que usan esos datos. Los conceptos básicos de este modelo son los siguientes:

- **Elemento:** es un campo de datos. Ejemplo: DNI.
- **Agregados de datos:** conjunto de datos con nombre. Ejemplo: Fecha (día, mes, año).
- **Tipos de registro:** representa un nodo, un conjunto de campos. Cada campo contiene elementos. Es la unidad básica de acceso y manipulación. Se asemeja a los registros en archivos o las entidades en el modelo E-R.
- **Conjunto:** colección de dos o más tipos de registro que establece una vinculación entre ellos. Uno de ellos se llama propietario y el otro, miembro. Tienen una relación muchos a muchos (M:M), que para representarla se necesita un registro conector. Los conjuntos están formados por un solo registro propietario y uno o más registros miembros. Un registro propietario no puede ser a la vez miembro de sí mismo. Una ocurrencia del conjunto está formada por un registro propietario y el resto son registros miembros. Una ocurrencia de registro no puede pertenecer a varias ocurrencias del mismo conjunto.
- **Ciclo:** se forma cuando un registro miembro tiene como descendientes a uno de sus antepasados.
- **Bucle, lazo o loop:** es un ciclo en el que los registros propietarios y miembros son el mismo.



Diagramas de estructura de datos en un modelo en red

El **diagrama** es el esquema que representa el diseño de una BD de red. Se basa en representaciones entre registros por medio de enlaces. Existen relaciones en las que participan solo dos entidades (binarias) y relaciones en las que participan más de dos entidades (generales) ya sea con o sin atributo descriptivo en la relación. Se utilizan cuadros o celdas para representar los registros y líneas para representar los enlaces.

Una ocurrencia del esquema son los valores que toman los elementos del esquema en un determinado momento. El modelo es muy flexible pues no hay restricciones. Esto implica la gran dificultad a la hora de implementarlo físicamente a la larga es poco eficiente. Para representar físicamente este modelo se pueden usar punteros y listas. Este modelo es sólo teórico, a la hora de llevarlo a la práctica se introducen las restricciones necesarias.

El diagrama de estructura de datos de red especifica la estructura lógica global de la BD; su representación gráfica se basa en la colocación de los campos de un registro en un conjunto de celdas que se ligan con otro(s) registro(s).

Vamos a suponer que tenemos una BD de red con datos de alumnos y asignaturas.

Registro ALUMNO:

```
struct ALUMNO
 { int NMatricula;
   char Nombre[35];
   int Curso;
   char Direccion[30];
 };
```

Registro ASIGNATURA:

```
struct ASIGNATURA
 { int Codigo;
   char NomAsig[35];
 };
```

Entre ALUMNOS y ASIGNATURA existe la relación de ALUMNO-CURSA-ASIGNATURA, que en el modelo E-R se representa como muestra la Figura 1.16:

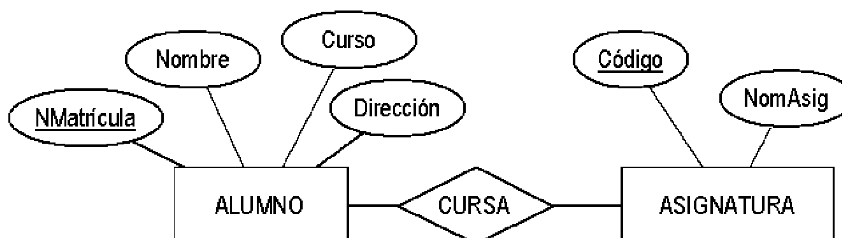


Figura 1.16 *Modelo E-R de ALUMNO-CURSA-ASIGNATURA.*

Las estructuras de datos según la cardinalidad se representan en los siguientes casos:

- **Representación del diagrama de estructura cuando el enlace o la relación no tiene atributos descriptivos**

  - *Caso 1*. Cardinalidad uno a uno (ver Figura 1.17).

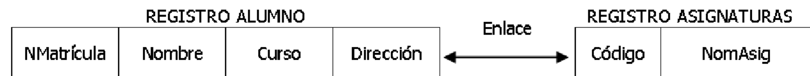


Figura 1.17. *Relación uno a uno sin atributos, en el modelo en red.*

  - *Caso 2*. Cardinalidad uno a muchos, un alumno cursa muchas asignaturas (ver Figura 1.18).

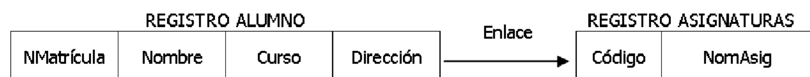


Figura 1.18. *Relación uno a muchos, en el modelo en red.*

  - *Caso 3*. Cardinalidad muchos a muchos, muchos alumnos cursan muchas asignaturas (ver Figura 1.19).

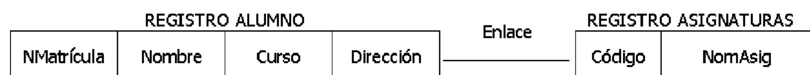


Figura 1.19. *Relación muchos a muchos en el modelo en red.*

En la Figura 1.20 se muestra un ejemplo de ocurrencia M:N:

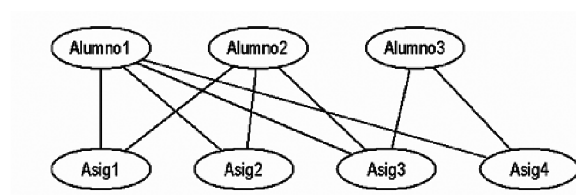


Figura 1.20. *Ejemplo de ocurrencias M:N en el modelo en red.*

- **Cuando el enlace tiene atributos descriptivos**

  Suponemos ahora que en la relación entre ALUMNOS y ASIGNATURAS necesitamos saber la Nota de un alumno en una asignatura, es decir, la relación lleva un atributo descriptivo. El diagrama del modelo E-R se muestra en la Figura 1.21:





Figura 1.21. *Modelo E-R de ALUMNO-CURSA-ASIGNATURA con el atributo NOTA.*

Para convertirlo al diagrama de estructura de datos realizaremos lo siguiente:

1. Se representan los campos del registro como antes.
2. Se crea un nuevo registro, que llamaremos NOTAS, con el campo Nota.
3. Se crean los enlaces indicando la cardinalidad. A los enlaces les daremos los nombres: ALUMNOTA para la relación entre ALUMNO y NOTAS, y ASIGNOTAS, para la relación entre ASIGNATURAS y NOTAS.

- **Dependiendo de la cardinalidad los diagramas de estructuras de datos se transforman como sigue**
  - *Caso 1*. Cardinalidad uno a uno. Un alumno cursa una asignatura con una sola nota, en la Figura 1.22 se representa el diagrama.

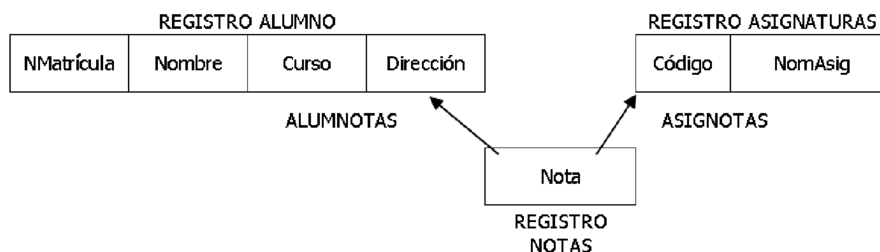


Figura 1.22. *Representación de la cardinalidad 1:1 en el modelo en red.*

  - *Caso 2*. Cardinalidad uno a muchos. Un alumno cursa muchas asignaturas que tienen una nota (ver Figura 1.23).

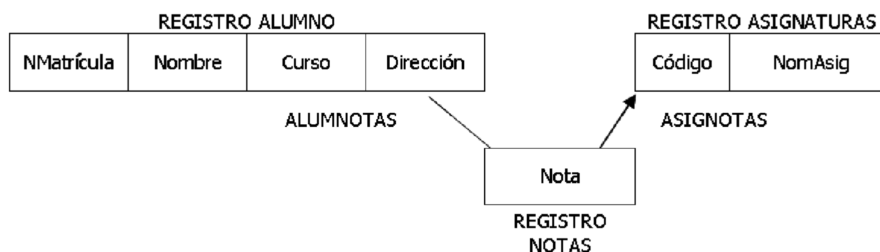


Figura 1.23. *Representación de la cardinalidad 1:M en el modelo en red.*

– *Caso 3*. Cardinalidad muchos a muchos. Muchos alumnos cursan muchas asignaturas con muchas notas (ver Figura 1.24).

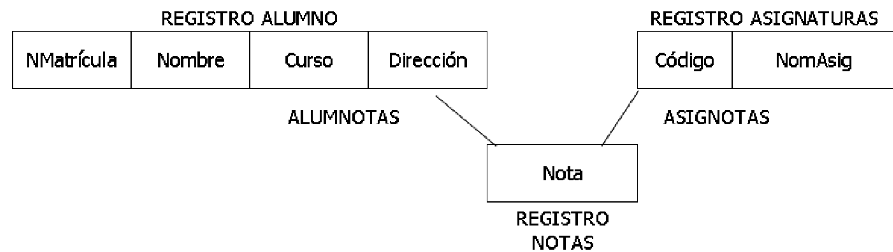


Figura 1.24. *Representación de la cardinalidad N:M en el modelo en red.*

- **Consideramos ahora que en la relación intervienen más de dos entidades y no hay atributos descriptivos en la relación.**

  Suponemos que nos interesa saber los profesores que imparten las asignaturas y agregamos a la relación ALUMNO-CURSA-ASIGNATURA la entidad PROFESOR. El diagrama E-R queda como se muestra en la Figura 1.25:

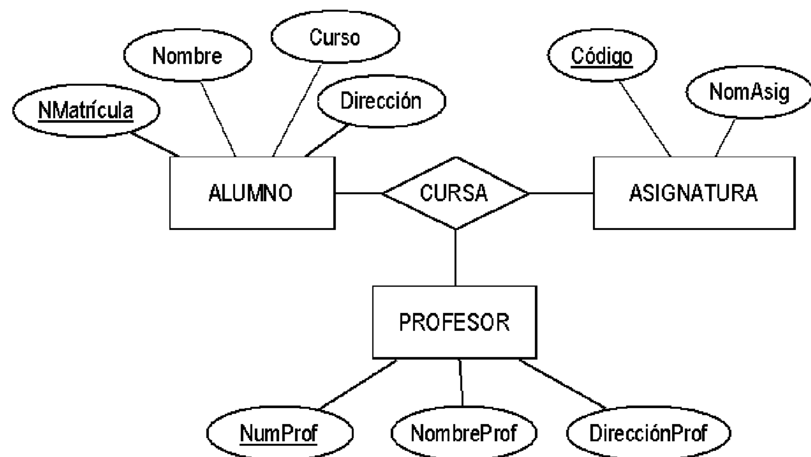


Figura 1.25. *Representación en el modelo E-R de la relación entre ALUMNO-ASIGNATURA-PROFESOR.*

Para realizar la transformación a diagramas de estructura de datos seguimos los siguientes pasos:

1. Se crean los registros para cada una de las entidades que intervienen.

2. Se crea un nuevo tipo de registro que llamaremos ENLACE, que puede no tener campos o tener sólo uno que contenga un identificador único, el identificador lo proporcionará el sistema y no lo utiliza directamente el programa de aplicación. A este registro se le denomina también como *registro de enlace* o *conector*.



– Considerando una relación con cardinalidad uno a uno, el diagrama de estructuras se muestra en la Figura 1.26:

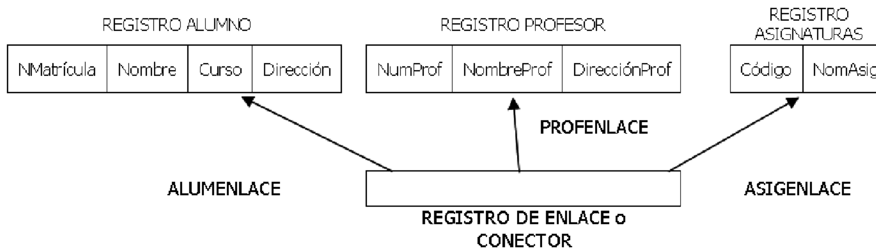


Figura 1.26. *Diagrama de estructuras de la relación uno a uno entre 3 entidades.*

Si el registro enlace tuviese atributos, estos se añadirán y se enlazarán indicando el tipo de cardinalidad de que se trate. Por ejemplo, añadimos el atributo Nota a la relación CURSA (ver Figura 1.27).

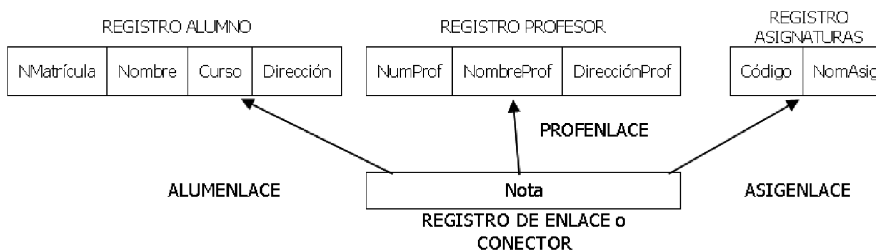


Figura 1.27. *Diagrama de estructuras de la relación uno a uno con registro de enlace y atributos.*

Si consideramos que un profesor imparte una sola asignatura y los alumnos tienen nota en una sola asignatura y esa asignatura sólo es cursada por un alumno, una instancia de este modelo sería la que se muestra en la Figura 1.28:



Figura 1.28. *Representación de una ocurrencia de la relación PROFESOR-ALUMNO-ASIGNATURA.*

El registro enlace contendrá los punteros para acceder a cada uno de los registros.

– Si ahora consideramos una cardinalidad muchos a muchos, es decir, un alumno cursa muchas asignaturas y el profesor imparte muchas asignaturas: una asignatura es impartida por muchos profesores y una asignatura es cursada por muchos alumnos, el diagrama de estructuras resultante se muestra en la Figura 1.29.

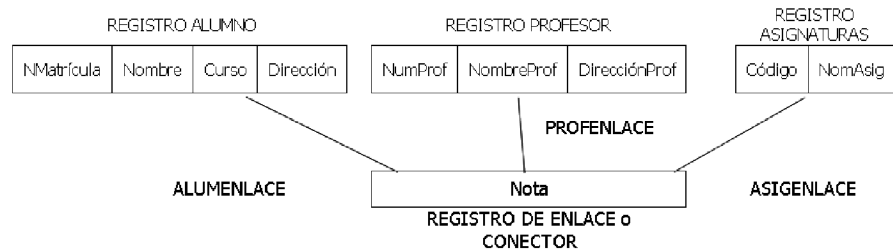


Figura 1.29. *Diagrama de estructuras para una relación M:M.*

Modelo de datos de CODASYL

Los SGDB que utilizan este modelo de red deben añadir restricciones a la hora de implementar físicamente la BD y obtener un mayor rendimiento del sistema. En 1971, un grupo conocido como CODASYL (*Conference on Data System Languages*) desarrolla el modelo DBTG (*Data Base Task Group*). Este grupo es el que desarrolló los estándares para COBOL. El modelo CODASYL ha evolucionado durante los últimos años y existen diversos productos SGBD orientados a transacciones, sin embargo actualmente estos productos se utilizan muy poco. El modelo es bastante complejo de implementar y los diseñadores y programadores deben de tener mucho cuidado al elaborar BBDD y aplicaciones DBTG. Además este modelo está enfocado al COBOL. Gran parte de las deficiencias detectadas son atribuye a que este modelo fue desarrollado antes de que se establecieran los conceptos esenciales de la tecnología de bases de datos. Ejemplos de BBDD en red son el DMS 1100 de UNIVAC, el EDMS de Xerox, el PHOLAS de Philips y el DBOMP de IBM.

En el modelo DBTG sólo pueden emplearse enlaces uno a uno y uno a muchos. En este modelo, existen dos elementos principales que son el propietario y el miembro, donde sólo puede existir un propietario y varios miembros, y cada miembro depende sólo de un propietario.

- **Conjuntos DBTG**

En este modelo sólo se utilizan enlaces uno a uno y uno a muchos. Podemos representar este modelo como se muestra en la Figura 1.30:



Figura 1.30. *Enlaces 1:1 y 1:M en el modelo DBTG. Conjunto DBTG.*

En el modelo DBTG esta estructura se denomina **conjunto DBTG** o **SET**. El nombre que se le asigna al conjunto suele ser el mismo que el de la relación que une a las entidades. Cada conjunto DBTG puede tener cualquier número de ocurrencias. Puesto que no se permiten enlaces del tipo muchos a muchos, cada ocurrencia del conjunto tiene exclusiva-



mente un propietario y cero o más registros miembros. Además, ningún registro puede participar en más de una ocurrencia del conjunto en ningún momento. Sin embargo, un registro miembro puede participar simultáneamente en varias ocurrencias de diferentes conjuntos. Añadir que un mismo registro no puede ser miembro y propietario a la vez, no está admitida la reflexividad.

En la Figura 1.31 se representan varias ocurrencias de un conjunto DBTG. La tercera ocurrencia no es válida pues todas pertenecen al mismo conjunto, y el miembro 3 no puede tener dos propietarios (ver Figura 1.31).

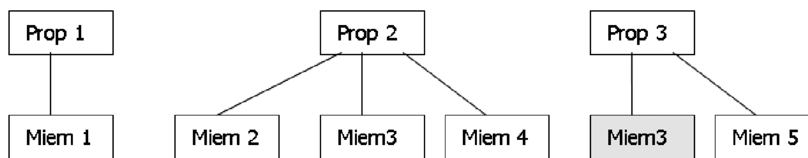


Figura 1.31. *Ocurrencias de un conjunto DBTG.*

Para declarar los conjuntos de un diagrama de estructuras como el que se muestra en la Figura 1.32 realizamos lo siguiente:

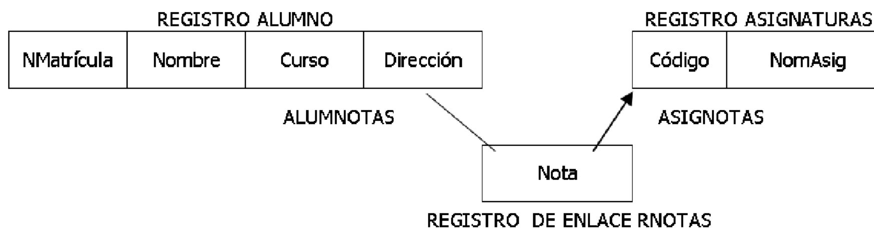


Figura 1.32. *Ejemplo diagrama de estructuras. Relación 1:M.*

En el diagrama existen dos conjuntos DBTG:

ALUMNOTAS, cuyo propietario es ALUMNO y cuyo miembro es RNOTAS.
ASIGNOTAS, cuyo PROPIETARIO es ASIGNATURAS y miembro RNOTAS.

La declaración de los conjuntos será:

```
Set name is ALUMNOTAS
Owner is ALUMNO
member is RNOTAS
```

```
Set name is ASIGNOTAS
Owner is ASIGNATURAS
member is RNOTAS
```

Al igual que otros modelos de datos, el DBTG proporciona un lenguaje de comandos para la manipulación de datos, así como para la actualización y el procesamiento de conjuntos DBTG.

1.7 Modelo jerárquico

El modelo jerárquico es similar al modelo de red. Los datos y las relaciones se representan mediante registros y enlaces. Se diferencia del modelo de red en que los registros están organizados como colecciones de árboles. El modelo jerárquico se sirve de árboles para la representación lógica de los datos, su implementación se lleva a cabo mediante árboles y punteros.

No se ha llegado a una formalización matemática del modelo y de sus lenguajes, como ha ocurrido en el caso del modelo relacional; tampoco se ha intentado su estandarización, como en el caso del DBTG. Los primeros sistemas de BBDD jerárquicas aparecieron en 1968: el IMS/360 (*Information Management System*) con su lenguaje de datos el DL/I de IBM, y el SYSTEM 2000 de Intel. Los productos basados en este tipo de modelos han perdido las altas cuotas de mercado de las que disfrutaban hace una década y se consideran sistemas muy superados por la tecnología relacional, aunque aún persisten muchas aplicaciones basadas en este modelo.

Características del modelo jerárquico

La representación gráfica del modelo jerárquico se realiza mediante un árbol invertido, en el que el nivel superior está formado por una única entidad o segmento bajo el cual cuelgan el resto de entidades o segmentos en niveles que se van ramificando. Los diferentes niveles quedan unidos por medio de las relaciones. El nivel más alto de la jerarquía tiene un nodo que se llama *raíz*. Cada nodo representa un tipo de registro llamado segmento con sus correspondientes campos. Los *segmentos* se organizan de manera que en un mismo nivel están todos aquellos que dependen de un segmento de nivel inmediatamente superior.

Los segmentos, en función de su situación en el árbol, se denominan:

- **Segmento padre:** es el que tiene descendientes, todos ellos localizados en el mismo nivel.
- **Segmento hijo:** es el que depende de un segmento de nivel superior. Los hijos de un mismo padre están en el mismo nivel.
- **Segmento raíz:** es el padre que no tiene padre. Ocupa el nivel superior del árbol. El segmento raíz es único.

Por ejemplo, tenemos la BD de un centro escolar llamada CENTROESCOLAR, que cuenta con 5 segmentos:

– El segmento raíz almacena los datos de los CURSOS que se imparten en el centro por ejemplo: Código, Descripción, Tipo enseñanza, Turno y Nivel.

– En el segundo nivel del árbol hay tres segmentos dependientes del segmento raíz. El primero de ellos contiene los datos de los PROFESORES que imparten clase en el curso: Código profesor, Nombre profesor, Dirección, Tlf y Especialidad. El segundo contiene los datos de los ALUMNOS que están matriculados en el curso, por ejemplo:



Número de matrícula, Nombre alumno, Dirección, Tlf, Fecha nacimiento. Y el tercero contiene los datos de las ASIGNATURAS que se imparten en el curso: Código, Descripción y Tipo asignatura.

– El tercer nivel del árbol está ocupado por un solo segmento que depende del segmento ASIGNATURAS y que contiene los datos de las AULAS donde se imparten las asignaturas: Código de aula, Localización y Tipo de aula. En la Figura 1.33. se muestra el esquema jerárquico de estos 5 segmentos a 3 niveles.

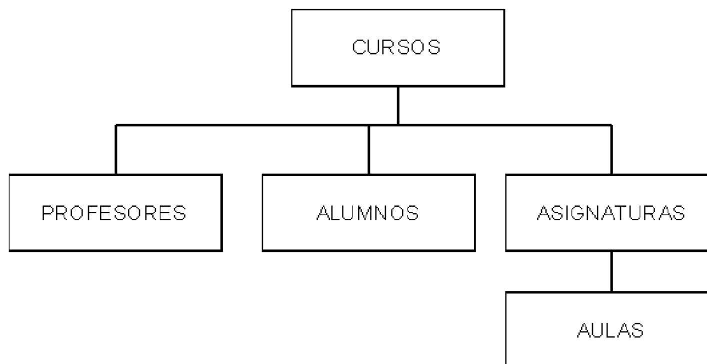


Figura 1.33. *Esquema jerárquico de 5 segmentos y 3 niveles.*

Definiciones del modelo jerárquico:

- Una OCURRENCIA de un segmento de una BD jerárquica es el conjunto de valores que toman todos los campos que lo componen en un momento determinado.
- Un REGISTRO es el conjunto formado por una ocurrencia del segmento raíz y todas las ocurrencias del resto de los segmentos de la BD que dependen jerárquicamente de dicha ocurrencia raíz.
- La relación PADRE/HIJO en la que se apoyan las BBDD jerárquicas, determina que el camino de acceso a los datos sea ÚNICO; este camino, denominado CAMINO SECUENCIA JERÁRQUICA, comienza siempre en una ocurrencia del segmento raíz y recorre la BD de arriba abajo, de izquierda a derecha y, por último, de adelante atrás.

Una estructura jerárquica, tiene las siguientes características:

– El árbol se organiza en un conjunto de niveles.

– El nodo raíz, el más alto de la jerarquía, se corresponde con el nivel 0.

– Las líneas que unen los nodos se llaman camino, no tienen nombre, ya que entre dos conjuntos de datos sólo puede haber una interrelación.

– Al nodo de nivel inferior sólo le puede corresponder un único nodo de nivel superior, es decir, un padre puede tener varios hijos y un hijo sólo tiene un padre.

- Todo nodo, a excepción del nodo raíz, ha de tener obligatoriamente un padre.
- Se llaman **hojas** a los nodos que no tienen hijos.
- Se llama **altura** al número de niveles de la estructura jerárquica.
- Se denomina **momento** al número de nodos.
- El número de hojas del árbol se llama **peso**.
- Sólo están permitidas las interrelaciones 1:1 ó 1:N
- Cada nodo no terminal y sus descendientes forman un subárbol, de forma que un árbol es una estructura recursiva.
- La suma total de un nodo padre y sus hijos se llama **familia**.



Actividades propuestas

4 **Dado el árbol que se muestra en la Figura 1.34, escribe sus características: los niveles, el número de hojas, altura, peso, recorrido y momento.**

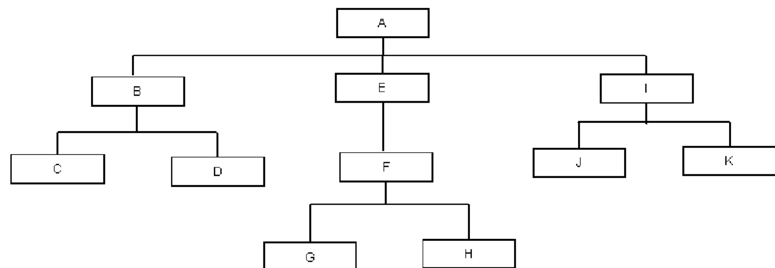


Figura 1.34. *Árbol jerárquico para la Actividad 4.*

Para acceder a la información de las BBDD jerárquicas se hace un recorrido ordenado del árbol. Se suele recorrer en preorden; es decir, raíz, subárbol izquierdo y subárbol derecho.

Las restricciones que presenta este modelo respecto a otros son las siguientes:

- Cada árbol debe tener un único segmento raíz.
- No puede definirse más de una relación entre dos segmentos dentro de un árbol.



- No se permiten las relaciones reflexivas de un segmento consigo mismo.
- No se permiten las relaciones N:M, esto implica la repetición de registros o la redundancias de datos.
- No se permite que exista un hijo con más de un padre.
- Para cualquier acceso a la información almacenada, es obligatorio el acceso por la raíz del árbol, excepto en el caso de utilizar un índice secundario.
- El árbol debe recorrer, siempre de acuerdo a un orden prefijado, el camino jerárquico.
- La estructura del árbol, una vez creada, no se puede modificar. La actualización de la estructura de la BD es bastante complicada, no se podrá insertar un nodo hijo si aún no tiene asignado un padre. La baja de un registro implica que desaparezca todo el subárbol que tiene dicho registro como nodo raíz.

Transformación de un esquema E-R en un esquema jerárquico

Interrelaciones 1:N con cardinalidad mínima 1 en la entidad padre: en este caso, no hay problemas. La representación es igual que en el modelo E-R. En la Figura 1.35 se representa una relación PROFESOR-ALUMNO. Un profesor puede tener muchos alumnos, y un alumno sólo pertenece a un profesor.

Interrelaciones 1:N con cardinalidad mínima 0 en el registro padre, en este caso pueden existir hijos sin padre, por lo que se crea un padre ficticio o se crean dos estructuras arborescentes. La primera estructura arborescente tendrá como nodo padre el registro PROFESOR, y como nodo hijo los identificadores del registro ALUMNO (la clave). De esta forma, no se introducen redundancias estando los atributos de ALUMNO en la segunda arborescencia en la cual sólo existe un nodo raíz ALUMNO sin descendientes. Ver Figura 1.35:

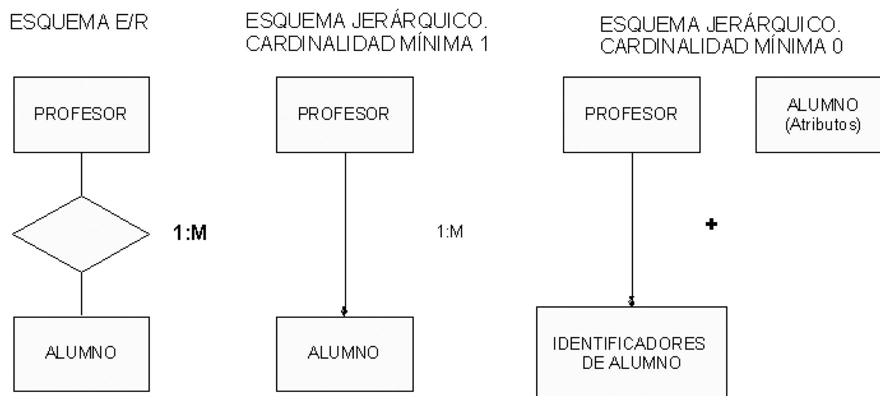


Figura 1.35. *Representación de relaciones 1:M en el modelo jerárquico.*

Si representamos una ocurrencia del segmento padre PROFESOR, observamos que cada profesor tendrá un número de ocurrencias de segmentos hijos, es decir, su propio árbol. Podemos afirmar que una BD jerárquica está formada por un conjunto de árboles disjuntos. Existirá un árbol para cada profesor. Ver Figura 1.36.

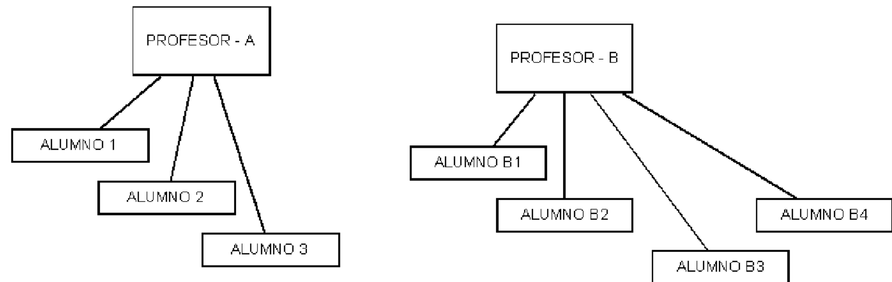
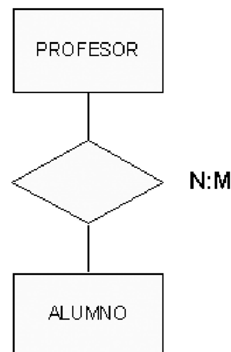


Figura 1.36. *Representación de ocurrencias del segmento PROFESOR.*

- **Interrelaciones N:M.** La solución es parecida a la anterior. Se crean dos arborescencias, en una de ellas se representa el nodo padre PROFESOR con los identificadores del nodo ALUMNO, y en el otro árbol se crean el nodo ALUMNO, sus atributos, relacionados con los identificadores del nodo padre PROFESOR. Ver Figura 1.37:



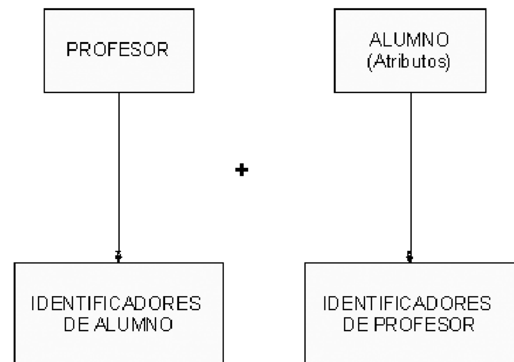


Figura 1.37. *Representación de relaciones M:N en el modelo jerárquico.*

- **Interrelaciones reflexivas.** En la Figura 1.38 se muestra la interrelación de PIEZA-FORMA PARTE DE-PIEZA, que representa a una PIEZA de un sistema de fabricación de productos de automóviles, por ejemplo, que forma parte de otra PIEZA. Una pieza es componente de otras piezas y está compuesta a su vez de otras piezas. Es una relación reflexiva M:N. Se utiliza la jerarquía de la izquierda si se desea obtener la explosión, es decir, obtener las piezas que forman parte de una pieza, y de la derecha para la implosión, es decir, obtener a partir de una pieza aquellas de las que forma parte.



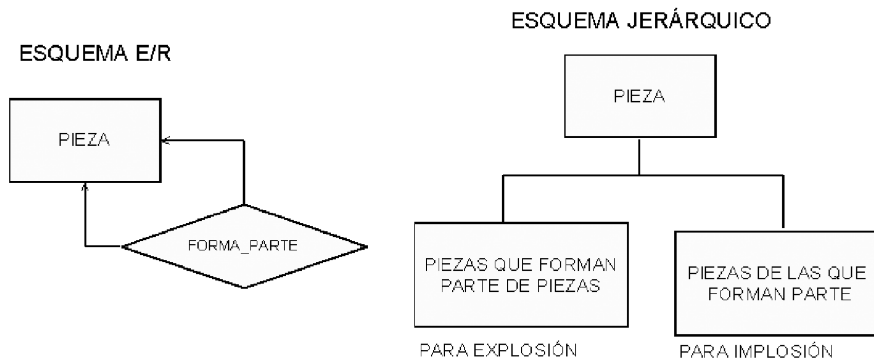


Figura 1.38. *Representación de relaciones reflexivas en el modelo jerárquico.*

La aplicación de todas estas normas de diseño evita la introducción de redundancias y la pérdida de simetría del árbol, sin embargo, complica el esquema jerárquico resultante pues estará formado por muchos árboles.

Actividades propuestas



5 **Representa el modelo jerárquico del diagrama E-R que se muestra en la Figura 1.39. Representa también una ocurrencia.**

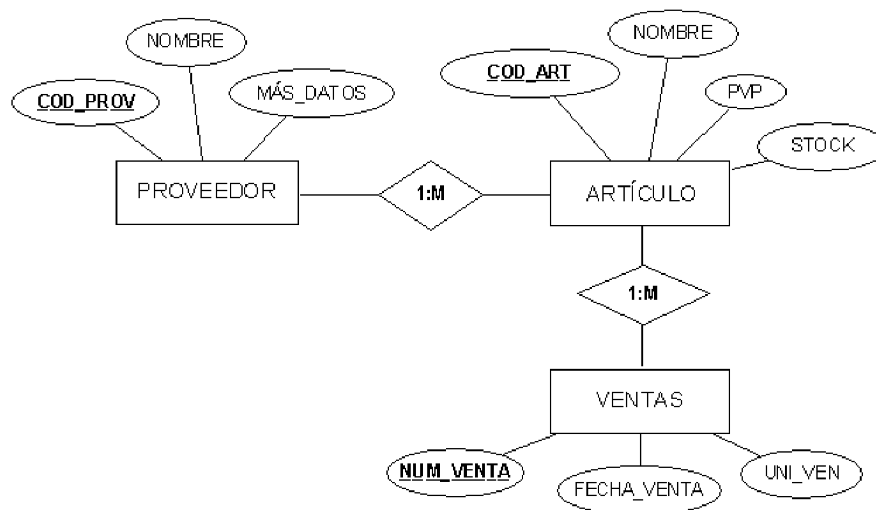


Figura 1.39. *Diagrama E-R para la Actividad.*

1.8 Modelo orientado a objetos

El modelo de datos orientado a objetos surge por las limitaciones del modelo relacional, sobre todo a la hora de abordar tipos de datos más complejos, y por la falta de capacidad semántica del modelo relacional para desarrollar aplicaciones en áreas como el diseño asistido por ordenador, la ingeniería del software, los sistemas basados en el conocimiento y el tratamiento de documentos, multimedia y gestión de redes, que requieren modelar objetos e interrelaciones más complejas.

Este modelo está basado en el paradigma de la programación orientada a objetos (POO). Los SGBO (Sistemas de Gestión de Bases de Objetos) o SDBDOO (Sistemas de Gestión de Bases de Datos Orientados a Objetos) gestionan objetos en los cuales están encapsulados los datos y las operaciones que interactúan con ellos. Además proporcionan un modelo único de datos. No hay diferencia entre el modelo conceptual (el modelo E-R) y el modelo lógico (el relacional), y las aplicaciones pueden acceder directamente al modelo.

Las SGBDOO adoptan una arquitectura que consta de un sistema de gestión que soporta un lenguaje de BBDD orientado a objetos, con una sintaxis similar a un lenguaje de programación también orientado a objetos, como puede ser C++ o Java. El lenguaje de BBDD es especificado mediante un lenguaje de definición de datos (ODL), un lenguaje de manipulación de datos (OML), y un lenguaje de consulta (OQL), siendo todos ellos portables a otros sistemas.

Estructura de los objetos

A nivel conceptual un objeto va a ser lo que una entidad en el modelo E-R, a la hora de implementarlo el objeto es un encapsulamiento de un conjunto de operaciones: código y datos en una única unidad, cuyo contenido no es visible desde el exterior. Esta ocultación de la información permite modificar aspectos del objeto sin que afecte a los demás que interactúan con él.

Las interacciones entre un objeto y el resto del sistema se realizan mediante un conjunto de mensajes.

Un objeto está formado por:

- Un conjunto de propiedades o atributos que contienen los datos del objeto, serían los atributos del modelo E-R.
- Un conjunto de mensajes a los que el objeto responde.
- Un conjunto de métodos, que es código para implementar los mensajes. Un método devuelve un valor como respuesta al mensaje.

En la Figura 1.40 podemos ver la representación de un objeto ALUMNO.



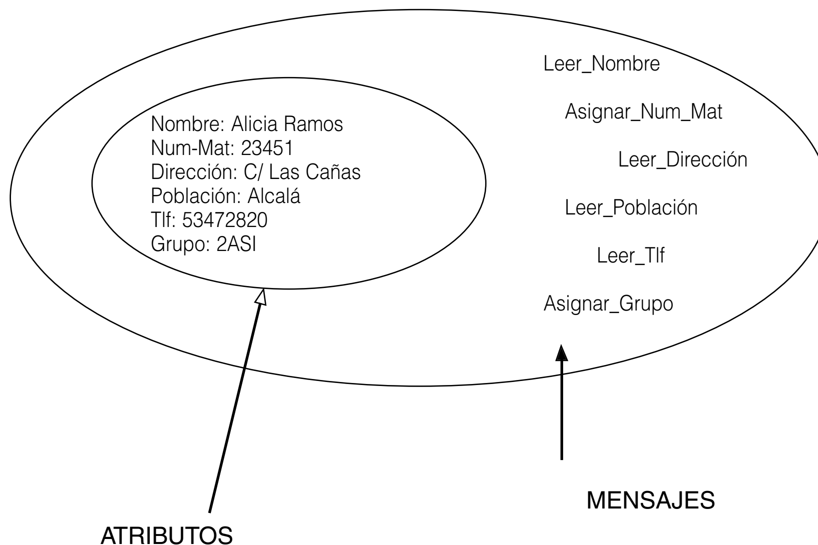


Figura 1.40. *Representación del objeto ALUMNO.*

Jerarquía de clases

Una **clase** es un conjunto de objetos similares. A cada uno de estos objetos se le llama *instancia de su clase*. Todos los objetos de la clase comparten una definición común y se comportan de la misma forma, aunque difieran en los valores asignados a los atributos. Podemos hacer una equivalencia entre entidad del modelo E-R y clase del modelo OO.

A continuación, se muestra la clase EMPLEADO en la que vemos los atributos y los mensajes:

```
class empleado
{
/* atributos */
string nombre;
string dirección;
date fecha_alta;
int salario;
/* mensajes */
string leer_nombre();
string leer_dirección();
date leer_fecha_alta();
int leer_salario();
};
```

Los métodos para el manejo de los mensajes no se han incluido en la clase.

Herencia

Las clases, en un sistema orientado por objetos, se representan en forma jerárquica como en las jerarquías de generalización en el modelo E-R. Hay superclases y subclases. La subclase hereda todos los atributos, mensajes y métodos de la superclase. Suponemos ahora que dentro de la clase EMPLEADO se consideran subclases oficinistas y transportistas. Cada una de ellas heredará los atributos y mensajes de la superclase, y además tendrá otros atributos y otros mensajes. Para identificarlas llevan la palabra clave `isa`. Ver el ejemplo que se muestra:

```
class oficinista isa empleado
{
/* atributos */
string especialidad;
int pulsaciones;
/* mensajes */
string leer_especialidad();
int calculo_pulsaciones(int pulsa);
};
class transportista isa empleado
{
/* atributos */
string tipo_carnet;
int horas_acumuladas;
/* mensajes */
string leer_tipo_carnet();
int sumar_horas(int horas);
};
```

Hay dos enfoques para la creación de BDOO: extender los SGBDR relacionales para que sean capaces de soportar los conceptos de la programación orientada a objetos, añadiendo a los lenguajes de BBDD existentes, como es el caso del SQL3, tipos complejos de datos y la programación orientada a objetos (los SGBD que proporcionan extensiones orientadas a objetos a los sistemas relacionales se denominan Bases de datos relacionales orientadas a objetos). Otra opción es que soporten un modelo de objetos puro y no estar basados en extensiones.

Éstas cogen un lenguaje de POO existente y se amplía para que trabaje con las bases de datos.

Polimorfismo

Otra característica importante del paradigma de la orientación a objetos es el polimorfismo, es decir, la capacidad de que un mensaje sea interpretado de formas distintas según el objeto que lo recibe. El polimorfismo es conocido como la sobrecarga de operadores. Este concepto permite enlazar el mismo nombre o símbolo de operador a dos o más implementaciones diferentes del operador, dependiendo del tipo de objetos a los que éste se aplique.



El modelo de datos orientado a objetos

Los SGBDOO soportan un modelo de objetos puro, en la medida en que no están basados en extensiones de otros modelos como el relacional. Están influenciados por los lenguajes de POO. Ejemplos de SGBDOO son Poet, Jasmine, ObjectStore y GemStone. En las BDOO, la organización ODMG (*Object Data Management Group*), que representa el 100% de las BDOO industriales, ha establecido un estándar que intenta definir un SGBDOO que integre las capacidades de las BBDD con las capacidades de los lenguajes de programación orientados a objetos, de forma que los objetos de la BD aparezcan como objetos del lenguaje de programación. De esta manera, intentan eliminar la falta de correspondencia existente entre los sistemas de tipos de ambos lenguajes. El ODMG define:

- Un modelo de objetos estándar para el diseño de estas BBDD.
- Lenguaje de definición de objetos (ODL, *Object Definition Language*)
- Lenguaje de consultas (OQL, Object Query Language), estilo al SQL. Vinculación con los lenguajes C++, Java y Smalltalk. Definen un lenguaje de manipulación de objetos (OML-*Object Manipulation Language*) que extiende el lenguaje de programación para soportar objetos persistentes (cualidad de algunos objetos de mantener su identidad y relaciones con otros objetos con independencia del sistema o proceso que los creó). El objeto persistente es el que tiene vida aunque el programa que trabaja con él haya terminado, es decir, los datos permanecen después de la sesión del usuario y la ejecución del programa de la aplicación. Para que esto sea así el objeto será guardado en una BD. A los lenguajes que incorporan estas estructuras de datos se les llama *lenguajes persistentes*.

Podemos considerar los SGBDOO como gestores de 3.ª generación. También están incluidos las BD objeto-relacionales. La primera generación serían los SGBD jerárquicos y en red, que utilizaban estructuras de datos tipo listas y árboles para su implementación, y la 2.ª los SGBDR relacionales, que utilizan estructuras tipo tabla (ver Figura 1.41).

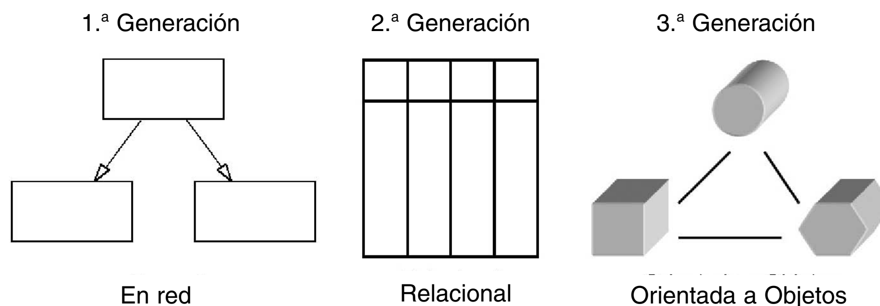


Figura 1.41. *Generaciones de modelos de BBDD.*

En la Tabla 1.3 se muestran los estándares y las BBDD más características de tercera generación:

BD OBJETO-RELACIONAL	BD OBJETOS PUROS
Estándar	Estándar
SQL: 1999 (SQL3), Melton 1999 SQL: 2003 , Melton 2003	ODMG-93, Cattell (1994), Cattell (1995) ODMG V.2.0, Cattell (1997) ODMG V.3.0, Cattell (2000)
Productos	Productos
Oracle (a partir de la V8) POSTGRES Universal Server de INFORMIX.	ObjectStore de Object Design. Persistencia de objetos en C++ y Java. POET. Persistencia de objetos en C++ y Java. Gemstone de Servi Logia, Meier y Stone (1987). Persistencia de objetos Smalltalk, soporta también C++ y Java.

Tabla 1.3. *Sistemas gestores de BBDD de 3.ª generación.*

Para representar el modelado de una BDOO se utilizan el diagrama de clases (que muestra la estructura de clases del sistema incluyendo las relaciones que puedan existir entre ellas) y el diagrama de objetos (que muestra un conjunto de objetos y sus relaciones en un momento determinado, equivale a lo que sería una instancia del diagrama de clases). Para realizar los modelados se utiliza el, **Lenguaje de Modelado Unificado** (*Unified Modeling Language-UML*). Es un lenguaje de modelado orientado a objetos que proporciona las técnicas para desarrollo de sistemas orientados a objetos. Es un modelado visual y utiliza diagramas para representar los esquemas del sistema.

Una clase se representa con un rectángulo dividido en tres secciones, mostrando en la primera el nombre de la clase (y, opcionalmente, otra información que afecte a la clase), en la segunda los atributos y en la última las operaciones. A continuación, se muestra la clase Empleado en la Figura 1.42.

Empleado
nombre dirección fecha_alta salario
leer_nombre() leer_dirección() leer_fecha_alta()

Figura 1.42. *Clase empleado.*

Las relaciones en UML son conexiones semánticas entre clases. Existen cuatro tipos principales de relaciones en UML:

- **Generalización:** es una relación de especialización (Padre-hijo o superclase-subclase). La generalización se utiliza para modelar la herencia en los lenguajes orientados a objetos. Una de las características de la herencia es que permite simplificar la construcción de clases relacionadas, ya que gracias a ella es posible agrupar las características comunes de un conjunto de clases en una clase padre (superclase) y hacer que todas ellas hereden de la superclase. En el ejemplo se muestra la generalización de la clase Empleado, ver Figura 1.43:



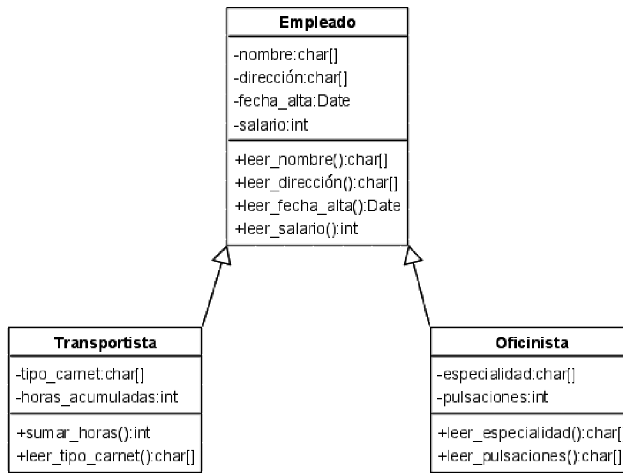


Figura 1.43. *Generalización de la clase Empleado.*

- **Asociación:** representan las relaciones entre instancias de clase. Se representa gráficamente como una línea que conecta las clases relacionadas. Si la clase Documento (A) se relaciona con la clase Persona (B), lo puede hacer de dos formas: en la primera, la clase A puede acceder a los atributos y operaciones públicas de la clase B y viceversa. Se representa con una línea, pero si añadimos una flecha que indique el sentido de dicha asociación restringimos la navegabilidad de la asociación. En el ejemplo, A puede acceder a los atributos y operaciones públicas de la clase B, pero B no puede acceder a A. Ver Figura 1.44:

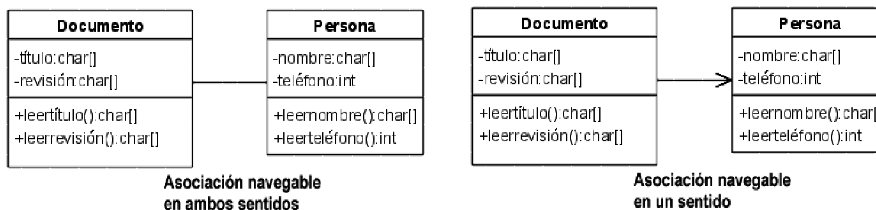


Figura 1.44. *Representación de asociaciones.*

También una clase puede relacionarse consigo misma, es decir, pueden modelarse asociaciones reflexivas.

En las asociaciones se puede añadir información para aumentar su expresividad y significado:

- **El nombre de la asociación,** un nombre descriptivo que indica la naturaleza de la asociación. Se añade un pequeño triángulo que indique el sentido en que se debe interpretar.

- **Roles,** para indicar el rol que desempeña cada una de las clases en la asociación. Se identifica por un nombre a los finales de la línea, describe la semántica de la relación en el sentido indicado. El rol puede estar representado en el nombre de la clase.

– **Multiplicidad,** describe la cardinalidad de la relación, es decir, cuántos objetos de una clase participan en la relación. Se representa por: 1 (relaciones 1-1), * (0-muchos), 0..1 (0-uno) , 1..* (1-muchos) y m..n (para especificar un número, por ejemplo 7...10).

En la figura que se muestra se representa una asociación entre las clases Documento y Persona, en la que indicamos el nombre de la asociación, los roles y la multiplicidad (ver Figura 1.45).

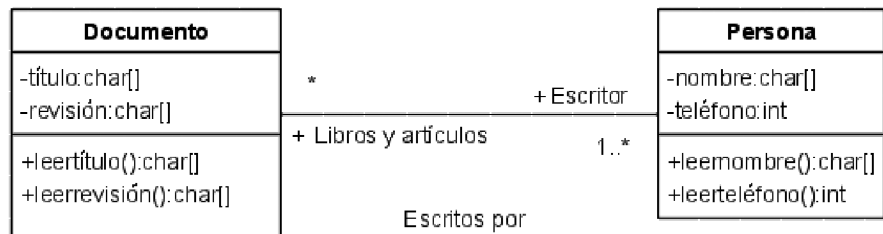


Figura 1.45. *Información de las asociaciones.*

- **Realización:** es una relación pactada y establecida entre clases, en la cual una clase especifica un contrato (por ejemplo, un interfaz) que la otra garantiza que cumplirá. Generalmente se emplea la realización para especificar una relación entre una interfaz y una clase o componente que implementa las operaciones especificadas en dicha interfaz.

  La interfaz especifica una serie de operaciones (cuya implementación estará en la clase) pero no proporciona ninguna implementación ni tiene atributos. Es equivalente a una clase abstracta que sólo tiene operaciones abstractas.

  Gráficamente, una interfaz se muestra como una clase con el estereotipo «interfaz» y sus operaciones, o en forma resumida como un círculo y debajo el nombre del interfaz.

  Para indicar que una clase realiza una interfaz, se muestra una línea discontinua acabada en flecha vacía que va de la clase que implementa la interfaz a la interfaz. Ver Figura 1.46:

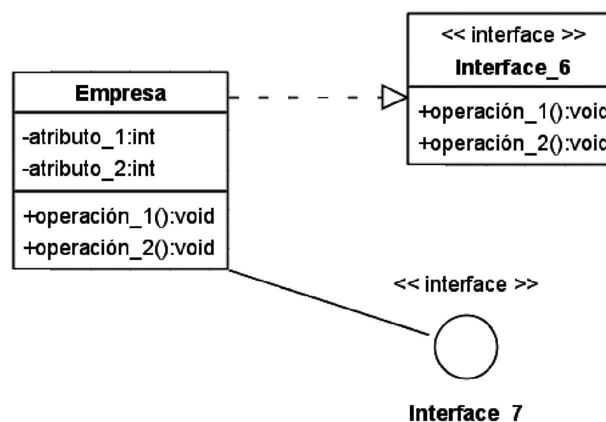


Figura 1.46. *Representación de una clase que realiza una interfaz.*



Cuando una clase realiza una interfaz, debe implementar todas las operaciones indicadas en la misma, además de las exclusivas de la clase. Las interfaces pueden evitar, en ciertos casos, el uso de herencia múltiple. Son muy útiles para independizar la implementación de sistemas o utilidades de los servicios u operaciones que proporcionan, fomentando la reutilización. Otra de las posibilidades de las interfaces consiste en limitar el acceso en las asociaciones entre clases. En el ejemplo que se muestra en la Figura 1.47 indicamos una asociación entre la clase Empresa y Persona a través de la interfaz Trabajador, que contiene los métodos que más interesan a la empresa. Esto evita que la empresa actúe con los datos médicos.

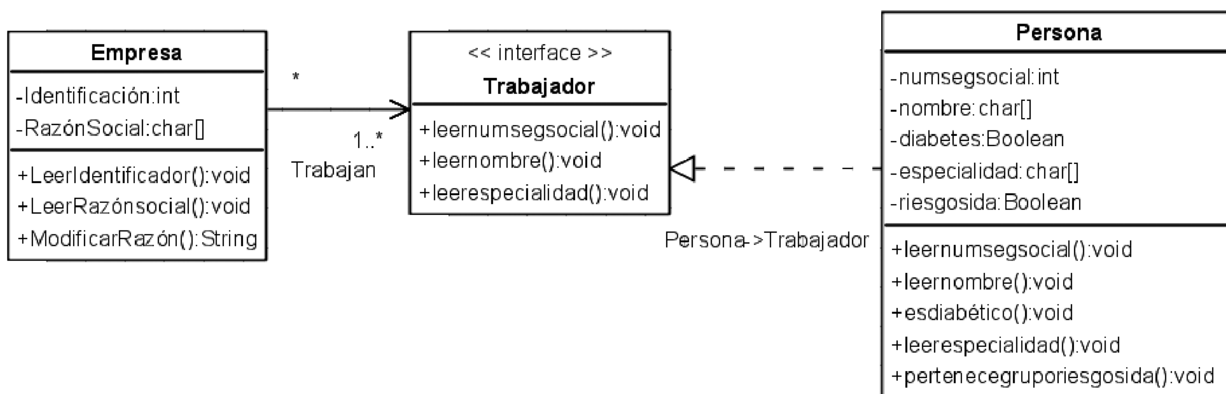


Figura 1.47. *Asociación a través de una interfaz.*

- **Dependencia:** es una relación entre entidades independientes y dependientes. El elemento dependiente (el cliente) requiere conocer al elemento independiente (el que proporciona el servicio) y que esté presente. Las dependencias se usan para modelar relaciones en las cuales un cambio en el elemento independiente (el suministrador) puede requerir cambios en el elemento dependiente (el cliente). Es un tipo de relación unidireccional, ya que el elemento dependiente debe conocer al independiente, pero el independiente desconoce la existencia del elemento dependiente. Gráficamente se muestra como una línea discontinua acabada en flecha que va del elemento dependiente al independiente (esto es, el elemento dependiente señala al elemento del que depende).

  Por ejemplo: la clase Cliente (dependiente) dependerá de servicios proporcionados por la clase Servidor (independiente), y un cambio en la clase Servidor puede ocasionar que la clase Cliente necesite ser adaptada. Un cambio en la clase Cliente no tiene ningún efecto sobre la clase Servidor, ya que esta última es independiente de la primera (no la necesita).

  Estas relaciones se utilizan cuando una clase usa los servicios de otra a través de una interfaz, es decir, depende de la interfaz. Un cambio en la interfaz requeriría un cambio en la clase dependiente. También se utilizan para mostrar la dependencia entre paquetes ya que las clases se suelen agrupar en paquetes.

Conceptos básicos



SGBD o DBMS (Sistema Gestor de Bases de Datos): colección de datos estructurados, organizados y relacionados entre sí, y el conjunto de programas que acceden y gestionan esos datos.

ANSI-SPARC (*American National Standard Institute - Standards Planning and Requirements Committee*): comité que propuso una arquitectura de tres niveles para los SGBD cuyo objetivo principal es el de separar los programas de aplicación de la BD física.

Diccionario de datos: es el lugar dónde se deposita información acerca de todos los datos que forman la BD. Contiene las características lógicas de los sitios donde se almacenan los datos del sistema, incluyendo nombre, descripción, alias, contenido y organización. Identifica también los procesos donde se emplean los datos y los sitios donde se necesita el acceso inmediato a la información.

Modelos de datos: el instrumento principal para ofrecer la abstracción de los datos, se utilizan para la representación y el tratamiento de los problemas y los representan a tres niveles de abstracción. En la Figura 1.48 podemos ver la relación entre los modelos lógicos y conceptuales de datos.

DBTG (*Data Base Task Group*): modelo de datos lógico de red creado por CODASYL.

ODMG (*Object Data Management Group*): organización que propone los estándares para definir SGBDOO.

Objetos persistentes: cualidad de algunos objetos para mantener su identidad y relaciones con otros objetos, aunque el programa que trabaja con él haya terminado.

Lenguajes persistentes: Son los lenguajes que incorporan objetos persistentes en sus estructuras de datos y permiten que el objeto sea almacenado en la BD.

Lenguaje de Modelado Unificado (*Unified Modeling Language*): lenguaje de modelado orientado a objetos que proporciona las técnicas para desarrollo de sistemas orientados a objetos.

Objetos persistentes: Cualidad de algunos objetos de mantener su identidad y relaciones con otros objetos, aunque el programa que trabaja con él haya terminado.

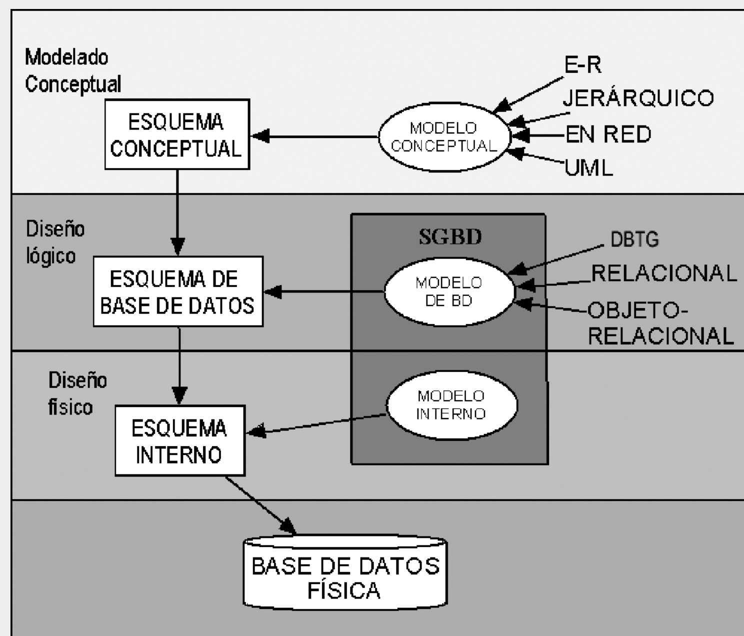


Figura 1.48. *Relación entre los modelos lógicos y conceptuales de datos.*





Actividades complementarias



1 **Realiza el diagrama de datos en el modelo E-R del enunciado:**
En una estación de autobuses contamos con unos autobuses que recorren una serie de lugares y que son conducidos por varios conductores. Se quiere representar los lugares que son recorridos por cada autobús, conducidos por cada conductor y la fecha en la que se visita el lugar. Define las entidades, los posibles atributos; identifica los atributos clave y los datos importantes para la relación entre las entidades. Indica también la cardinalidad.

2 **Realiza el diagrama de datos del ejercicio anterior en el modelo en red.**

3 **Realiza el diagrama de datos en el modelo E-R, que represente el siguiente enunciado:**
La Consejería de Educación gestiona varios tipos de centros: públicos, privados y concertados. Los privados tienen un atributo específico que es la cuota y los concertados la agrupación y la comisión. También asigna plazas a los profesores de la comunidad para impartir clase en esos centros. Un profesor puede impartir clase en varios centros.
Define las entidades, los posibles atributos; identifica los atributos clave, y los datos importantes para la relación entre las entidades. Indica también la cardinalidad.

4 **Representa el ejercicio anterior en el modelo O.O. Indica las posibles operaciones. Considera las asociaciones navegables en ambos sentidos. Indica el nombre de la asociación, los roles, la multiplicidad y las generalizaciones.**

5 **Realiza el diagrama de datos en el modelo E-R, que represente este problema:**
A un taller de automóviles llegan clientes a comprar coches. De los coches nos interesa saber la marca, el modelo, el color y el número de bastidor.
Los coches pueden ser nuevos y de segunda mano. De los nuevos nos interesa saber las unidades que hay en el taller. De los viejos el año de fabricación, el número de averías y la matrícula.
Los mecánicos se encargan de poner a punto los coches usados del taller. Un mecánico pone a punto a varios coches usados.
Un cliente puede comprar varios coches; un coche puede ser comprado por varios clientes. De la compra nos interesa la fecha y el precio.
Define las entidades, los atributos, las relaciones, sus atributos si los hubiera y las cardinalidades.

6 **Representa el ejercicio anterior en el modelo O.O. Considera las asociaciones navegables en ambos sentidos. Indica el nombre de la asociación, los roles y la multiplicidad.**

7 **Realiza el diagrama de datos en el modelo E-R que represente el siguiente problema:**
Una agencia de viajes está formada por varias oficinas que se ocupan de atender a los posibles viajeros. Cada oficina oferta un gran número de viajes. Los viajes trabajan con una serie de destinos y una serie de procedencias. Cada viaje tiene un único destino y una única procedencia. Sin embargo, un destino puede ser objetivo de varios viajes y una procedencia ser punto de partida de varios viajes. Cada viaje tiene muchos viajeros.

8 **Realiza el diagrama de datos en el modelo E-R de este enunciado:**
Una entidad bancaria está formada por varias sucursales y cada sucursal tiene un gran número de cuentas que son propiedad de los clientes. Los datos saldo, debe y haber deben aparecer en cada una de las cuentas. Las cuentas son de dos tipos: cuenta de ahorro con el atributo específico tipo de interés. Y cuenta corriente, con el atributo específico cantidad de descubierto. Las cuentas o son corrientes o son de ahorro.
Un cliente puede tener varias cuentas. Una cuenta es sólo propiedad de un cliente.
Las cuentas realizan una serie de movimientos, en los que además de otros datos deben aparecer la cantidad implicada y la fecha. Existe una serie de tipos de movimientos reconocidos por el banco. Un movimiento pertenece a un tipo. Sin embargo, de un tipo de movimiento puede haber varios movimientos.

El modelo de datos relacional

2

En esta unidad aprenderás a:

1. Describir la estructura del modelo de datos relacional.
2. Transformar el modelo E-R al modelo lógico relacional.
3. Diseñar bases de datos relacionales.
4. Normalizar esquemas relacionales.
5. Realizar operaciones básicas sobre tablas utilizando álgebra relacional.





2.1 Introducción

Ya se ha visto en la Unidad anterior lo que es un *sistema gestor de bases de datos*, y cuáles son sus objetivos. También se vio la *arquitectura ANSI* para su aplicación en la creación de las bases de datos. Se han estudiado distintos modelos conceptuales de datos pero no se ha llegado a realizar ningún diseño lógico. Esta Unidad trata del que actualmente es el principal modelo para las aplicaciones de procesamiento de datos, el *modelo relacional,* que presenta una forma muy simple y potente de representar los datos.

A lo largo de la Unidad, se exponen los fundamentos del modelo de datos relacional y su aplicación para el diseño lógico de datos y de bases de datos relacionales.

2.2 El modelo relacional

El modelo de datos relacional fue desarrollado por E.F. Codd para IBM, a finales de los años sesenta. Propone un modelo basado en la teoría matemática de las relaciones, con el objetivo de mantener la independencia de la estructura lógica respecto al modo de almacenamiento y otras características de tipo físico. El modelo de Codd persigue, al igual que la mayoría de los modelos de datos, los siguientes objetivos:

- **Independencia física de los datos.** El modo de almacenamiento de los datos no debe influir en su manipulación lógica.
- **Independencia lógica de los datos.** Los cambios que se realicen en los objetos de la base de datos no deben repercutir en los programas y usuarios que acceden a la misma.
- **Flexibilidad.** Para presentar a los usuarios los datos de la forma más adecuada a la aplicación que utilicen.
- **Uniformidad** en la presentación de las estructuras lógicas de los datos, que son tablas, lo que facilita la concepción y manipulación de la base de datos por parte de los usuarios.
- **Sencillez.** Pues las características anteriores, así como unos lenguajes de usuario sencillos, hacen que este modelo sea fácil de comprender y utilizar por el usuario.

Para conseguir estos objetivos, Codd introduce el concepto de *relación* (tabla) como estructura básica del modelo. Todos los datos de una base de datos se representan en forma de relaciones cuyo contenido varía en el tiempo. El modelo relacional se basa en dos ramas de las matemáticas: la *teoría de conjuntos* y la *lógica de predicados*. Esto hace que sea un modelo seguro y robusto.

En 1985, Codd publica sus famosas doce reglas analizando algunos de los productos comerciales de la época, que debe cumplir cualquier base de datos para ser considerada relacional:

– **Regla de información.** Toda información de una base de datos relacional está representada mediante valores en tablas.

- **Regla de acceso garantizado.** Se garantiza que todos los datos de una base relacional son lógicamente accesibles a través de una combinación de nombre de tabla, valor de clave primaria y nombre de columna.

- **Tratamiento sistemático de valores nulos.** Los valores nulos se soportan en los SGBD para representar la falta de información de un modo sistemático e independiente de los tipos de datos.

- **Catálogo en línea dinámico basado en el modelo relacional.** La descripción de la base de datos se representa en el ámbito lógico de la misma forma que los datos ordinarios, de modo que los usuarios autorizados pueden acceder a ellos utilizando el mismo lenguaje relacional.

- **Regla de sublenguaje completo de datos.** Un sistema relacional puede soportar varios lenguajes y varios modos de uso terminal. Sin embargo, debe haber al menos un lenguaje cuyas sentencias se puedan expresar mediante alguna sintaxis bien definida, como cadenas de caracteres, y que ofrezca completamente todos los puntos siguientes:

  - Definición de datos.
  - Definición de vistas.
  - Manipulación de datos (interactiva y por programa).
  - Restricciones de integridad.
  - Autorización.
  - Gestión de transacciones (comienzo, confirmación y vuelta atrás).

- **Regla de actualización de vista.** Todas las vistas, que sean teóricamente actualizables, son también actualizables por el sistema.

- **Inserción, actualización y supresión de alto nivel.** La capacidad de manejar una relación de base de datos o una relación derivada como un único operando se aplica no solamente a la recuperación de datos, sino también a la inserción, actualización y supresión de los datos.

- **Independencia física de los datos.** Los cambios que se efectúan tanto en la representación del almacenamiento, como en los métodos de acceso no deben afectar ni a los programas de aplicación ni a las actividades con los datos.

- **Independencia lógica de los datos.** Del mismo modo, los cambios que se efectúen sobre las tablas de la base de datos no modifican ni a los programas ni a las actividades con los datos.

- **Independencia de la integridad.** Las restricciones de integridad específicas de una base de datos relacional deben ser definidas mediante el sublenguaje de datos relacional y almacenarse en el catálogo de la base de datos.

- **Independencia de la distribución.** Un SGBD es independiente de la distribución.





- **Regla de no subversión.** Si un SGBDR tiene un lenguaje de bajo nivel (una fila cada vez) no se puede utilizar para destruir o evitar las reglas de integridad o las restricciones expresadas en el lenguaje relacional de alto nivel (varias filas al mismo tiempo).

El modelo relacional propone una representación de la información que:

- Origine esquemas que representen fielmente la información, los objetos y las relaciones existentes entre ellos forman el dominio del problema.
- Sea fácilmente entendida por los usuarios.
- Sea posible ampliar el esquema de la base de datos sin modificar la estructura lógica existente y los programas de aplicación.
- Permita la máxima flexibilidad en la formulación de los interrogantes sobre la información mantenida en la base de datos.

2.3 Estructura del modelo relacional

Como ya se ha indicado anteriormente, la **relación** es el elemento básico del modelo relacional y se representa como una tabla, en la que se puede distinguir el nombre de la tabla, el conjunto de columnas que representan las propiedades de la tabla y que se les llama *atributos*, y el conjunto de filas llamadas *tuplas*, que contienen los valores que toma cada uno de los atributos para cada elemento de la relación.

Una relación tiene una serie de elementos característicos que la distinguen de una tabla:

- No admiten filas duplicadas.
- Las filas y columnas no están ordenadas.
- La tabla es plana. En el cruce de una fila y una columna sólo puede haber un valor, no se admiten atributos multivaluados.

En la figura que se muestra a continuación se representa una relación llamada ALUMNO en forma de tabla.

RELACIÓN ALUMNOS

NUM_MAT	NOMBRE	APELLIDOS	CURSO
5467	JUAN	CABELLO	1BACH-A
3421	DOLORES	GARCÍA	1BACH-C
7622	JESÚS	SÁNCHEZ	2BACH-C

ATRIBUTOS
TUPLAS

Figura 2.1. *Representación de una relación en forma de tabla.*

A continuación, se exponen los elementos que constituyen el modelo relacional.

A. Dominios y atributos

Se define **dominio** como el conjunto finito de valores homogéneos (todos del mismo tipo) y atómicos (son indivisibles), que puede tomar cada atributo. Los valores contenidos en una columna pertenecen a un dominio que previamente se ha definido. Todos los dominios tienen un nombre y un tipo de datos asociado. Existen dos tipos de dominios:

- **Dominios generales.** Son aquellos cuyos valores están comprendidos entre un máximo y un mínimo. Por ejemplo, el Código_postal, que está formado por todos los números enteros positivos de 5 cifras.

- **Dominios restringidos.** Son los que pertenecen a un conjunto de valores específico. Por ejemplo, Sexo, que puede tomar los valores H o M.

Se define **atributo** como el papel o rol que desempeña un dominio en una relación. Representa el uso de un dominio para una determinada relación. El atributo aporta un significado semántico a un dominio. Por ejemplo, en la relación ALUMNO podemos considerar los siguientes dominios:

- Atributo NUM_MAT, dominio: conjunto de enteros formados por 4 dígitos.

- Atributo NOMBRE, dominio: conjunto de 15 caracteres.

- Atributo APELLIDOS, dominio: conjunto de 20 caracteres.

- Atributo CURSO, dominio: conjunto de 7 caracteres.

B. Relaciones

La relación se representa mediante una tabla con filas y columnas. Un SGBD sólo necesita que el usuario pueda percibir la base de datos como un conjunto de tablas. Esta percepción sólo se aplica a la estructura lógica de la base de datos (nivel externo y conceptual de la arquitectura a tres niveles ANSI-SPARC). No se aplica a la estructura física de la base de datos, que se puede implementar con distintas estructuras de almacenamiento.

En el modelo relacional, las relaciones se utilizan para almacenar información sobre los objetos que se representan en la base de datos. Se representa gráficamente como una tabla bidimensional en la que las filas corresponden a registros individuales y las columnas a los campos o atributos de esos registros. La relación está formada por:

- **Atributo (columna).** Se trata de cada una de las columnas de la tabla. Las columnas tienen un nombre y pueden guardar un conjunto de valores. Una columna se identifica siempre por su nombre, nunca por su posición. El orden de las columnas en una tabla es irrelevante.

- **Tupla (fila).** Representa una fila de la tabla. En la Figura 2.2 aparece la tabla EMPLEADO con dos filas o tuplas.





De las tablas se derivan los siguientes conceptos:

- **Cardinalidad.** Es el número de filas de la tabla. En el ejemplo anterior es dos.

- **Grado.** Es el número de columnas de la tabla. En el ejemplo anterior el grado es cinco.

- **Valor.** Viene representado por la intersección entre una fila y una columna. Por ejemplo, son valores de la tabla EMPLEADO: 13407877B, Milagros Suela Sarro, 1500.

- **Valor Null.** Representa la ausencia de información.

Nº Emple	Apellidos	Salario	Numdepart	FechaAlta
13407877B	Milagros Suela Sarro	1.500	10	18/11/90
41667891C	José María Cabello	2.000	20	29/10/92

Figura 2.2. *Tabla EMPLEADO con dos filas.*

Definiciones formales

La definición matemática de una **relación entre *n* dominios** (D1, D2, … Dn) es un subconjunto del producto cartesiano de estos dominios donde cada elemento de la relación, la tupla, es una serie de *n* valores ordenados. Una relación R definida sobre un conjunto de dominios D1, D2, … Dn consta de:

- **Cabecera:** conjunto fijo de pares atributo:dominio, {(A1: D1), (A2: D2), …….. (An: Dn) }, donde cada atributo corresponde a un único dominio y todos los atributos son distintos. El grado de la relación R es *n*.

- **Cuerpo:** conjunto variable de tuplas. Cada tupla es un conjunto de pares atributo: valor, {(A1: vi1), (A2: vi2), (An: vim) }, con i=1,2,….m, donde *m* representa la cardinalidad de la relación R. En cada par (Ai: vij) se tiene que el valor pertenece al dominio, vij Dj.

Para establecer la cabecera de la relación EMPLEADO primero se definen los dominios, que tienen un nombre y un tipo de datos asociado:

- Dominio NUME_EMPLE, conjunto de 9 caracteres, de los cuales 8 son dígitos y el último es una letra.

- Dominio NOMBRES, conjunto de 25 caracteres.

- Dominio PAGA, conjunto de 4 dígitos.

- Dominio DEPART, posibles valores de números de departamento, rango de 01 a 99.

- Dominio FECHAS, conjunto de 10 caracteres.

La cabecera de la tabla EMPLEADO sería:

```
{(NºEmple:NUME_EMPLE),(Apellidos:NOMBRES),(Salario:PAGA)
, (Numdepart:DEPART),(FechaAlta:FECHAS)}
```

Una de las tuplas es:

```
{(NºEmple: 13407877B ),(Apellidos: Milagros Suela Sarro),
(Salario: 1500), (Numdepart:10),(FechaAlta: 18/11/1990)}
```

Propiedades de las relaciones

Las relaciones tienen las siguientes características:

- Cada relación tiene un nombre y éste es distinto de los demás.
- Los valores de los atributos son *atómicos*: en cada tupla cada atributo toma un solo valor. Se dice que las relaciones están normalizadas.
- No hay dos atributos que se llamen igual.
- El orden de los atributos es irrelevante, no están ordenados.
- Cada tupla es distinta de las demás, no hay tuplas duplicadas.
- Al igual que en los atributos, el orden de las tuplas es irrelevante, las tuplas no están ordenadas.

Tipos de relaciones

En un SGBD relacional pueden existir varios tipos de relaciones, aunque no todos manejan todos los tipos. Unas relaciones permanecen en la base de datos y otras son los resultados de consultas:

- **Relaciones base.** Son relaciones reales que tienen nombre y forman parte directa de la base de datos almacenada. Se corresponde con el nivel conceptual de la arquitectura ANSI.
- **Vistas.** Se corresponde con el nivel externo de la arquitectura ANSI. Son relaciones con nombre que se definen a partir de una consulta. No tienen datos almacenados, lo que se almacena es la definición de la consulta. Se las llama también *virtuales*.
- **Instantáneas.** Se corresponde con el nivel interno de la arquitectura ANSI. Son relaciones con nombre y derivadas de otras. Son relaciones de sólo lectura y se refrescan periódicamente por el sistema.
- **Resultados de consultas.** Son las resultantes de las consultas de usuario. No persisten en la base de datos.





– **Resultados intermedios.** Son las relaciones que contienen los resultados de las subconsultas de usuario. No persisten en la base de datos.

– **Resultados temporales.** Son relaciones con nombre, similares a las relaciones base, pero se destruyen automáticamente en algún momento previamente determinado.

C. Claves

En una relación no hay tuplas repetidas. Se identifican de un modo único mediante los valores de sus atributos. Toda fila debe estar asociada con una clave que permita identificarla. A veces la fila se puede identificar por un único atributo, pero otras veces es necesario recurrir a más de un atributo. La clave debe cumplir dos requisitos:

– **Identificación unívoca.** En cada fila de la tabla el valor de la clave ha de identificarla de forma unívoca.

– **No redundancia.** No se puede descartar ningún atributo de la clave para identificar la fila.

Se define **clave candidata de una relación** como el conjunto de atributos que identifican unívoca y mínimamente (necesarios para identificar la tupla) cada tupla de la relación. Siempre hay una clave candidata pues por definición no puede haber dos tuplas iguales. Habrá un atributo o atributos que identifiquen la tupla.

Una relación puede tener más de una clave candidata entre las cuales se distinguen:

- **Clave primaria o principal** (*primary key*): aquella clave candidata que el usuario escoge para identificar las tuplas de la relación. No puede tener valores nulos. Si sólo existe una clave candidata, ésta se elegirá como clave primaria.

- **Clave alternativa:** aquellas claves candidatas que no han sido escogidas como clave primaria.

Se denomina **clave ajena de una relación R1** al conjunto de atributos cuyos valores han de coincidir con los valores de la clave primaria de otra relación R2. Ambas claves estarán definidas sobre el mismo dominio y son muy importantes en el estudio de la integridad de datos del modelo relacional.

Caso práctico



1 **Disponemos de las tablas TDEPART y TEMPLE. Las columnas de la tabla TDEPART son:**

Nº. de departamento (NUMDEPT), Nombre de departamento (NOMDEPT), Presupuesto (PRESUPUESTO).

Las claves candidatas son Nº. de departamento y Nombre de departamento, pues son únicos y no se van a repetir. Podemos escoger como primaria el Nº. de departamento. Normalmente se elegirán como claves primarias los campos códigos o número de empleado, departamento o artículo, que suelen ser numéricos.

(Continúa)

(Continuación)

Las columnas de la tabla TEMPLE son:

Nº. de empleado (NUMEMP), Apellido (APELLIDO), Nº. de departamento (NUMDEP), Salario (SALARIO).

La clave candidata es el Nº. de empleado. El apellido se puede repetir, así que no se la considera candidata. Como sólo hay una se escoge primaria el Nº. de empleado.

El atributo Nº. de departamento es clave ajena; relaciona las tablas TEMPLE y TDEPART.

La Figura 2.3 muestra el contenido de las tablas TEMPLE y TDEPART con sus claves primarias y ajenas.

TDEPART

CLAVE PRIMARIA		
NUMDEPT	**NOMDEPT**	**PRESUPUESTO**
D1	Marketing	1.000
D2	Desarrollo	1.200
D3	Investigación	5.000

TEMPLE

CLAVE PRIMARIA		CLAVE AJENA	
NUMEMP	**APELLIDO**	**NUMDEP**	**SALARIO**
E1	López	D1	1.500
E2	Fernández	D2	1.600
E3	Martínez	D3	1.800
E4	Sánchez	D2	2.000

Figura 2.3. *Tablas TEMPLE y TDEPART con las claves primarias y ajenas.*

D. Esquema de la base de datos

Una **base de datos relacional** es un conjunto de relaciones normalizadas. Para representar un esquema de una base de datos se debe dar el nombre de sus relaciones, los atributos de éstas, los dominios sobre los que se definen estos atributos, las claves primarias y las claves ajenas.

En el esquema los nombres de las relaciones aparecen seguidos de los nombres de los atributos encerrados entre paréntesis. Las claves primarias son los atributos subrayados, y las claves ajenas se representan mediante diagramas referenciales.

El esquema de la base de datos de empleados y departamentos es el siguiente:

TDEPART	(<u>NUMDEPT</u>, NOMDEPT, PRESUPUESTO)
TEMPLE	(<u>NUMEMP</u>, APELLIDO, NUMDEP, SALARIO)
TEMPLE —NUMDEP→ TEDEPART:	Departamento al que pertenece el empleado





2.4 Restricciones del modelo relacional

En todos los modelos de datos existen restricciones que a la hora de diseñar una base de datos se tienen que tener en cuenta. Los datos almacenados en la base de datos han de adaptarse a las estructuras impuestas por el modelo y deben cumplir una serie de reglas para garantizar que son correctos. El modelo relacional impone dos tipos de restricciones. Algunas de ellas ya las hemos citado en las propiedades de las relaciones y las claves. Los tipos de restricciones son:

- **Restricciones inherentes al modelo,** indican las características propias de una relación que han de cumplirse obligatoriamente y que diferencian una relación de una tabla. No hay dos tuplas iguales, el orden de las tuplas y los atributos no es relevante. Cada atributo sólo puede tomar un único valor del dominio al que pertenece. Ningún atributo que forme parte de la clave primaria de una relación puede tomar un valor nulo.

- **Restricciones semánticas o de usuario,** que representan la semántica del mundo real. Éstas hacen que las ocurrencias de los esquemas de la base de datos sean válidos. Los mecanismos que proporciona el modelo para este tipo de restricciones son los siguientes:

  a) La restricción de clave primaria (PRIMARY KEY) permite declarar uno o varios atributos como clave primaria de una relación.

  b) La restricción de unicidad (UNIQUE) permite definir claves alternativas. Los valores de los atributos no pueden repetirse.

  c) La restricción de obligatoriedad (NOT NULL) permite declarar si uno o varios atributos no pueden tomar valores nulos.

  d) Integridad referencial o restricción de clave ajena (FOREIGN KEY). Se utiliza para enlazar relaciones, mediante claves ajenas, de una base de datos. La integridad referencial indica que los valores de la clave ajena en la relación hijo se corresponden con los de la clave primaria en la relación padre.

  Además de definir las claves ajenas hay que tener en cuenta las operaciones de borrado y actualización que se realizan sobre las tuplas de la relación referenciada. Las posibilidades son las siguientes:

  - **Borrado y/o modificación en cascada** (CASCADE). El borrado o modificación de una tupla en la relación padre (relación con la clave primaria) ocasiona un borrado o modificación de las tuplas relacionadas en la relación hija (relación que contiene la clave ajena). En el caso de empleados y departamentos, si se borra un departamento de la tabla TDEPART se borrarán los empleados que pertenecen a ese departamento. Igualmente ocurrirá si se modifica el NUMDEPT de la tabla TDEPART esa modificación se arrastra a los empleados que pertenezcan a ese departamento.

  - **Borrado y/o modificación restringido** (RESTRICT). En este caso no es posible realizar el borrado o la modificación de las tuplas de la relación padre si existen tuplas relacionadas en la relación hija. Es decir, no podría borrar un departamento que tiene empleados.

- **Borrado y/o modificación con puesta a nulos** (SET NULL). Esta restricción permite poner la clave ajena en la tabla referenciada a NULL si se produce el borrado o modificación en la tabla primaria o padre. Así pues, si se borra un departamento, a los empleados de ese departamento se les asignará NULL en el atributo NUMDEPT.

- **Borrado y/o modificación con puesta a valor por defecto** (SET DEFAULT). En este caso, el valor que se pone en las claves ajenas de la tabla referenciada es un valor por defecto que se habrá especificado en la creación de la tabla.

e) Restricciones de verificación (CHECK). Esta restricción permite especificar condiciones que deban cumplir los valores de los atributos. Cada vez que se realice una inserción o una actualización de datos se comprueba si los valores cumplen la condición. Rechaza la operación si no se cumple.

A continuación, se muestran dos sentencias de creación de tablas, la tabla FABRICANTES (tabla maestra o padre) y la tabla ARTÍCULOS (tabla relacionada o hija). Un fabricante fabrica muchos artículos. Observa las restricciones:

```
CREATE TABLE FABRICANTES(
CD_FAB NUMBER(3) NOT NULL DEFAULT 100,
NOMBRE VARCHAR2(15) UNIQUE,
PAIS VARCHAR2(15)CONSTRAINT CK_PA
CHECK(PAIS=UPPER(PAIS)),
CONSTRAINT PK_FA PRIMARY KEY (CD_FAB),
CONSTRAINT CK_NO CHECK(NOMBRE=UPPER(NOMBRE))
);

CREATE TABLE ARTICULOS(
ARTIC  VARCHAR2(20)NOT NULL,
COD_FA NUMBER(3) NOT NULL,
PESO NUMBER(3) NOT NULL CONSTRAINT CK1_AR CHECK
(PESO>0),
CATEGORIA VARCHAR2(10) NOT NULL,
PRECIO_VENTA  NUMBER (4) CONSTRAINT CK2_AR CHECK
(PRECIO_VENTA>0),
PRECIO_COSTO  NUMBER (4) CONSTRAINT CK3_AR CHECK
(PRECIO_COSTO>0),
EXISTENCIAS NUMBER (5),
CONSTRAINT PK_ART PRIMARY KEY (ARTIC, COD_FA, PESO,
CATEGORIA),
CONSTRAINT FK_ARFA FOREIGN KEY (COD_FA) REFERENCES
FABRICANTES (CD_FAB) ON DELETE CASCADE,
CONSTRAINT CK_CAT CHECK(CATEGORIA
IN('Primera','Segunda', 'Tercera'))
);
```

f) Aserciones (ASSERTION). Son parecidas a la anterior, pero en este caso en lugar de afectar a una relación como CHECK, puede afectar a dos o más relaciones. La condición se establece sobre elementos de distintas relaciones. Pueden impli-





car a subconsultas en la condición. La definición de una aserción debe tener un nombre. Tiene vida por sí misma.

g) Disparadores (TRIGGER). Las restricciones anteriores son declarativas, sin embargo, este tipo es procedimental. El usuario podrá especificar una serie de acciones distintas ante una determinada condición. El usuario escribe el procedimiento a aplicar dependiendo del resultado de la condición. Los disparadores están soportados a partir de los estándares SQL3.

   A continuación, se muestra un disparador de base de datos que audita las operaciones de inserción y borrado de datos en la tabla EMPLE. Cada vez que se realiza una operación de actualización o borrado se inserta en la tabla AUDITAREMPLE una fila que contendrá: la fecha y hora de la operación, el número y apellido del empleado afectado, y la operación que se realiza. La creación de triggers se verá en las siguientes unidades.

```
CREATE OR REPLACE TRIGGER auditar_act_emp
 BEFORE INSERT OR DELETE ON EMPLE FOR EACH ROW
BEGIN
 IF DELETING THEN
  INSERT INTO AUDITAREMPLE
    VALUES(TO_CHAR(sysdate,'DD/MM/YY*HH24:MI*')
      ||  :OLD.EMP_NO||  '*'  ||  :OLD.APELLIDO  ||  '*
BORRADO ');
 ELSIF INSERTING THEN
    INSERT INTO AUDITAREMPLE
         VALUES(TO_CHAR(sysdate,'DD/MM/YY*HH24:MI*')
   || :NEW.EMP_NO || '*' || :NEW.APELLIDO||'* INSER-
CION ');
 END IF;
END;
```

2.5 Transformación de un esquema E-R a un esquema relacional

Una vez obtenido el esquema conceptual mediante el modelo E-R, hay que definir el modelo lógico de datos. Las reglas básicas para transformar un esquema conceptual E-R a un esquema relacional son las siguientes:

- Toda entidad se transforma en una tabla.
- Todo atributo se transforma en columna dentro de una tabla.
- El identificador único de la entidad se convierte en clave primaria.
- Toda relación N:M se transforma en una tabla que tendrá como clave primaria la concatenación de los atributos clave de las entidades que asocia.



Caso práctico

2 **Dado el esquema que se muestra en la Figura 2.4, en el que se representan las compras de artículos que hacen los clientes, convierte el esquema E-R en relacional.**

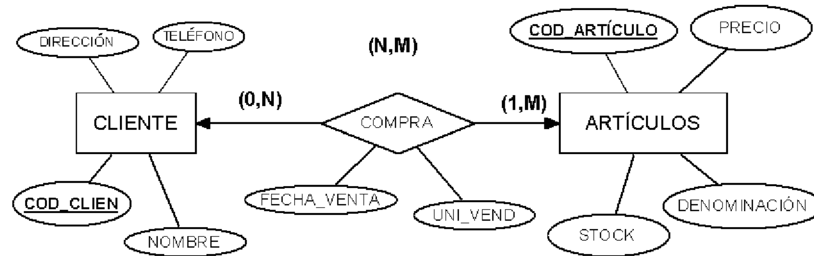


Figura 2.4. *Esquema E-R CLIENTE-COMPRA-ARTICULOS.*

1. Cada entidad se convierte en una tabla y los atributos de las entidades se convierten en columnas de las tablas. Lo representamos así:

   CLIENTE (**COD_CLIEN**, NOMBRE, DIRECCIÓN, TELEFONO)
   ARTICULOS (**COD_ARTICULO**, PRECIO, STOCK, DENOMINACIÓN)

2. La relación N:M se convierte a tabla. El nombre que se la da es el que pongamos a la relación, en este caso COMPRA. La clave estará formada por la concatenación de claves de las tablas anteriores. Estas a su vez pasan a ser claves ajenas y referencian a las tablas CLIENTE y ARTÍCULO.

   En esta relación, además de añadir las claves de las entidades anteriores, se añaden los atributos que intervienen en la relación. La tabla nos queda así:

   COMPRA (**COD_CLIEN (FK)**, **COD_ARTICULO(FK)**, UNI_VEND, FECHA_VENTA)

En la Figura 2.5 se muestra cómo quedaría gráficamente el esquema relacional creado en Access. Observa la ventana de relaciones en las que se indica las opciones de borrado y de modificación.

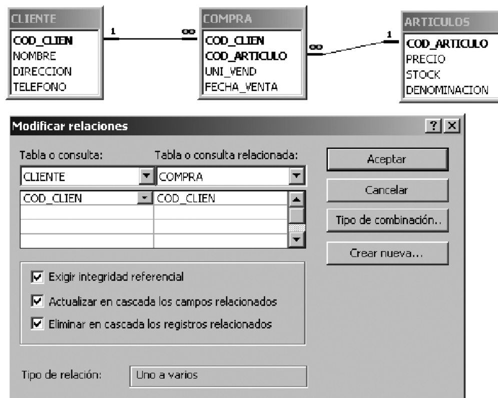


Figura 2.5. *Representación gráfica relacional de CLIENTE-COMPRA-ARTÍCULOS.*

(Continúa)





(Continuación)

Las opciones de borrado y de modificción suelen ser en cascada. A continuación, se muestra el código que crearía estas tres tablas:

```
CREATE TABLE CLIENTE (
  COD_CLIEN     NUMBER(6) NOT NULL PRIMARY KEY,
  NOMBRE        VARCHAR2(15),
  DIRECCION     VARCHAR2(15),
  TELEFONO      NUMBER(10));

CREATE TABLE ARTICULOS(
  COD_ARTICULO  NUMBER(6) NOT NULL PRIMARY KEY,
  PRECIO        NUMBER (6,2),
  STOCK         NUMBER (4),
  DENOMINACION  VARCHAR2(15));

CREATE TABLE COMPRA(
  COD_CLIEN     NUMBER(6) NOT NULL,
  COD_ARTICULO  NUMBER(6) NOT NULL,
  UNI_VEND      NUMBER (4),
  FECHA_VENTA   DATE,
  CONSTRAINT PK_COMPRA PRIMARY KEY (COD_CLIEN,COD_ARTICULO),
  CONSTRAINT FK_CLIEN FOREIGN KEY (COD_CLIEN)
                      REFERENCES CLIENTE (COD_CLIEN)
                      ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT FK_ARTIC FOREIGN KEY (COD_ARTICULO)
                      REFERENCES ARTICULOS (COD_ARTICULO)
                      ON DELETE CASCADE ON UPDATE CASCADE);
```

– En la transformación de relaciones 1:N existen dos soluciones:

- **Transformar la relación en una tabla.** Se hace como si se tratara de una relación N:M. Esta solución se realiza cuando se prevé que en un futuro la relación se convertirá en N:M y cuando la relación tiene atributos propios. También se crea una nueva tabla cuando la cardinalidad es opcional, es decir, (0,1) y (0,M). La clave de esta tabla es la de la entidad del lado *muchos*.

- **Propagar la clave.** Este caso se aplica cuando la cardinalidad es obligatoria, es decir, cuando tenemos cardinalidad (1,1) y (0,M) o (1,M). Se propaga el atributo principal de la entidad que tiene de cardinalidad máxima 1 a la que tiene de cardinalidad máxima N, desapareciendo el nombre de la relación. Si existen atributos propios en la relación, éstos también se propagarán.



Caso práctico

3 **Dado el esquema que se muestra en la Figura 2.6, en el que se representa la clasificación de los libros en temas, convertir el esquema E-R en relacional.**

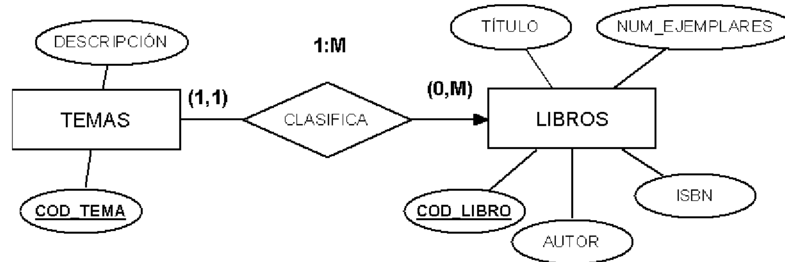


Figura 2.6. *Esquema E-R. TEMA-CLASIFICA-LIBRO.*

1. Se convierten a tabla las dos entidades:

   TEMAS (**COD_TEMA**, DESCRIPCIÓN)
   LIBROS (**COD_LIBRO**, AUTOR, ISBN, TÍTULO, NUM_EJEMPLARES)

2. Se propaga la clave, de la entidad TEMAS a la entidad LIBROS. La entidad LIBROS quedará así:

   LIBROS (**COD_LIBRO**, AUTOR, ISBN, TÍTULO, NUM_EJEMPLARES, **COD_TEMA(FK)**)

En la Figura 2.7 podemos observar cómo quedaría gráficamente el esquema relacional, creado en Access.

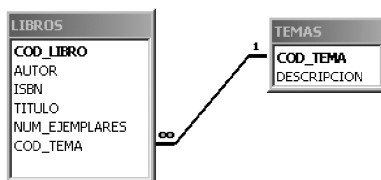


Figura 2.7. *Representación gráfica relacional de TEMA-CLASIFICA-LIBRO.*

– En la transformación de relaciones 1:1 se tienen en cuenta las cardinalidades de las entidades que participan. Existen dos soluciones:

- **Transformar la relación en una tabla.** Si las entidades poseen cardinalidades (0,1), la relación se convierte en una tabla.

- **Propagar la clave.** Si una de las entidades posee cardinalidad (0,1) y la otra (1,1), conviene propagar la clave de la entidad con cardinalidad (1,1) a la tabla resultante de la entidad de cardinalidad (0,1). Si ambas entidades poseen cardinalidades (1,1), se puede propagar la clave de cualquiera de ellas a la tabla resultante de la otra. En este caso, también se pueden añadir los atributos de una entidad a otra, resultando una única tabla con todos los atributos de las entidades y de la relación, si los hubiera, eligiendo como clave primaria una de las dos.





Caso práctico



4 **Consideramos la relación EMPLEADO-OCUPA-PUESTOTRABAJO. Un empleado ocupa un solo puesto de trabajo, y ese puesto de trabajo es ocupado por un solo empleado o por ninguno. El esquema E-R se muestra en la Figura 2.8; transformarlo al modelo relacional.**

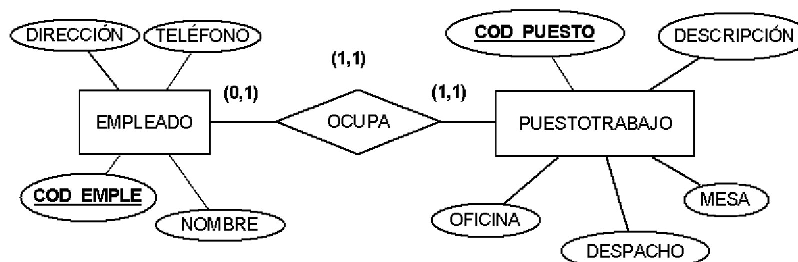


Figura 2.8. *Esquema E-R de EMPLEADO-OCUPA-PUESTOTRABAJO.*

En este caso, la clave se propaga desde la entidad PUESTOTRABAJO, con cardinalidad (1,1), a la entidad EMPLEADO, con cardinalidad (0,1). Las tablas las representaremos así:

PUESTOTRABAJO (**COD_PUESTO**, DESCRIPCIÓN, OFICINA, DESPACHO, MESA)
EMPLEAD (**COD_EMPLE**, NOMBRE, DIRECCIÓN, TELEFONO, **COD_PUESTO**(**FK**))

En la Figura 2.9 se representa gráficamente el esquema relacional, creado en Access. En este caso, hay que tener en cuenta que el COD_PUESTO de EMPLEADO no se puede repetir, por lo tanto llevará la restricción de UNIQUE, aparte de ser clave ajena.

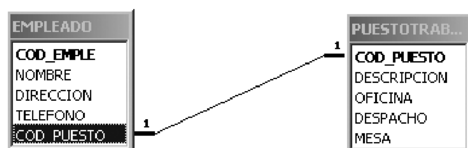


Figura 2.9. *Representación gráfica relacional de EMPLEADO-OCUPA-PUESTOTRABAJO.*

Actividades propuestas



1 **Realiza el diagrama E-R que cumpla las especificaciones y pásalo al modelo de datos relacional.**

Se desea mecanizar la biblioteca de un centro educativo. En la biblioteca existen fichas de autores y libros. Un autor puede escribir varios libros, y un libro puede ser escrito por varios autores. Un libro está formado por ejemplares que son los que se prestan a los usuarios.

Así un libro tiene muchos ejemplares y un ejemplar pertenece sólo a un libro. De los ejemplares nos interesa saber la localización dentro de la biblioteca. Los ejemplares son prestados a los usuarios, un usuario puede tomar prestados varios ejemplares y un ejemplar puede ser prestado a varios usuarios. Del préstamo nos interesa saber la fecha de préstamo y la de devolución.

A. Transformación de otros elementos del modelo E-R

- **Relaciones reflexivas o recursivas**

  Son relaciones binarias en las que participa un tipo de entidad. En el proceso de convertir una relación reflexiva a tabla hay que tener en cuenta sobre todo la cardinalidad. Lo normal es que toda relación reflexiva se convierta en dos tablas, una para la entidad y otra para la relación. Se pueden presentar los siguientes casos:

  – Si la relación es 1:1, la clave de la entidad se repite, con lo que la tabla resultante tendrá dos veces ese atributo, una como clave primaria y otra como clave ajena de ella misma. No se crea la segunda tabla.

  – Si la relación es 1:M, podemos tener dos casos:

    a) Caso de que la entidad muchos sea siempre obligatoria se procede como en el caso 1:1.

    b) Si no es obligatoria, se crea una nueva tabla cuya clave será la de la entidad del lado muchos, y además se propaga la clave a la nueva tabla como clave ajena.

Caso práctico

5 **Consideramos la relación EMPLEADO-DIRIGE-EMPLEADO. Un empleado puede dirigir a muchos empleados o a ninguno si es el que dirige. Y un empleado es dirigido por un director o por ninguno si el es el que dirige. En este caso no hay obligatoriedad en la entidad muchos. El esquema E-R se muestra en la figura 2.10, transformarlo al modelo relacional.**

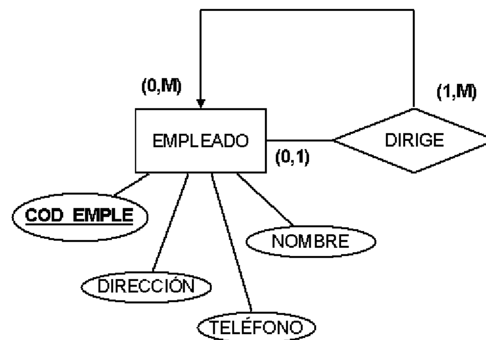


Figura 2.10. *Esquema E-R de EMPLEADO-DIRIGE-EMPLEADO.*

La tabla DIRIGE tiene como clave primaria el código de empleado, que a su vez será clave ajena. Además se le añade el código de empleado pero, en este caso, tendrá el papel de director, que a su vez será clave ajena a la tabla EMPLEADO. El resultado será el siguiente:

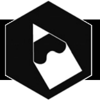
EMPLEADO (**COD_EMPLE**, DIRECCIÓN, TELÉFONO, NOMBRE)
DIRIGE (**COD_EMPLE(FK)**, **COD_DIREC(FK)**)





– Si es N:M, se trata igual que en las relaciones binarias. La tabla resultante de la relación contendrá dos veces la clave primaria de la entidad del lado muchos, más los atributos de la relación si los hubiera. La clave de esta nueva tabla será la combinación de las dos.

Caso práctico



6 **Consideramos la relación: una pieza se compone de muchas piezas, que a su vez están compuestas de otras piezas, es decir, PIEZA-COMPONE-PIEZA. El esquema E-R se muestra en la Figura 2.11. Transfórmalo al modelo relacional.**

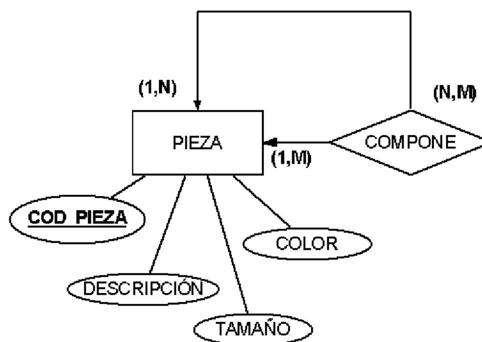


Figura 2.11. *Esquema E-R de PIEZA-COMPONE-PIEZA.*

En este caso, obtenemos la segunda tabla COMPONE_PIEZA en la que aparecerá repetido el código de pieza y formará la clave de la tabla. El atributo COD_PIEZ_CON representa la pieza que se compone de otras, y COD_PIEZA tiene el papel de pieza que compone a otras piezas. Ambos atributos son claves ajenas a la tabla PIEZA. La representación quedará así:

PIEZA (**<u>COD_PIEZA</u>**, DESCRIPCIÓN, TAMAÑO, COLOR)
COMPONE_PIEZA (**<u>COD_PIEZ_COM(FK)</u>**, **<u>COD_PIEZA(FK)</u>**)

- **Generalizaciones, transformación de jerarquías al modelo relacional**

El modelo relacional no dispone de mecanismos para la representación de las relaciones jerárquicas, así pues, las relaciones jerárquicas se tienen que eliminar. Para ello hay que tener en cuenta:

– La especialización que los subtipos tienen respecto a los supertipos, es decir, los atributos diferentes que tengan asociados cada uno de los subtipos, que son los que se diferencian con el resto de atributos de los otros subtipos.

– El tipo de especialización que representa el tipo de relación jerárquica: *total* o *parcial exclusiva*, y *total* o *parcial solapada*.

– Otros tipos de relación que mantengan tanto los subtipos como el supertipo.

– La forma en la que se va a acceder a la información que representan tanto el supertipo como el subtipo.

Teniendo en cuenta los puntos señalados para pasar estas relaciones al modelo relacional se aplicará una de las siguientes reglas:

A. **Integrar todas las entidades en una única eliminando a los subtipos.** Esta nueva entidad contendrá todos los atributos del supertipo, todos los de los subtipos, y los atributos discriminativos para distinguir a qué subentidad pertenece cada atributo. Todas las relaciones se mantienen con la nueva entidad. Esta regla puede aplicarse a cualquier tipo de jerarquía. La gran ventaja es la simplicidad pues todo se reduce a una entidad. El gran inconveniente es que se generan demasiados valores nulos en los atributos opcionales propios de cada entidad.

B. **Eliminación del supertipo,** transfiriendo los atributos del supertipo a cada uno de los subtipos. Las relaciones del supertipo se consideran para cada uno de los subtipos. La clave genérica del supertipo pasa a cada uno de los subtipos. Sólo puede ser aplicada para jerarquías totales y exclusivas. Inconvenientes que presenta esta transformación:

   – Se crea redundancia en la información, pues los atributos del supertipo se repiten en cada uno de los subtipos.

   – El número de relaciones aumenta, pues si el supertipo tiene relaciones, éstas pasan a cada uno de los subtipos.

C. **Insertar una relación 1:1 entre el supertipo y cada uno de los subtipos.** Los atributos se mantienen y cada subtipo se identificará con la clave ajena del supertipo. El supertipo mantendrá una relación 1:1 con cada subtipo. Los subtipos mantendrán, si la relación es exclusiva, la cardinalidad mínima 0, y si es solapada 0 ó 1.

Caso práctico

7 **Consideramos los profesores que imparten clases en dos tipos de centros educativos: públicos y privados. Un profesor puede impartir clase en varios centros, ya sean públicos o privados. Consideramos la asignatura como atributo de la relación entre profesor imparte en centro. Los centros educativos sólo pueden ser de estos dos tipos. Un centro público no puede ser a la vez privado. Los atributos específicos para los centros públicos son el presupuesto y los servicios, para los privados la organización y la cuota. La jerarquía a representar es total y exclusiva. En la Figura 2.12 se presenta el esquema E-R.**

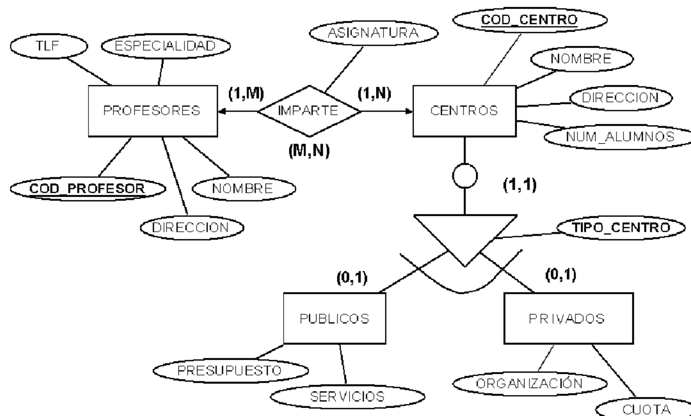


Figura 2.12. *Esquema E-R de PROFESORES-CENTROS.*

(Continúa)





(Continuación)

Observa que se ha añadido el atributo TIPO_CENTRO que no se ha dado en el enunciado, sin embargo en las generalizaciones conviene añadir un atributo que identifique el tipo del subtipo. Para la transformación podemos aplicar cualquiera de las tres reglas ya que se trata de una jerarquía total y exclusiva:

A. Integramos los subtipos en el supertipo. El esquema E-R será el que se muestra en la Figura 2.13:

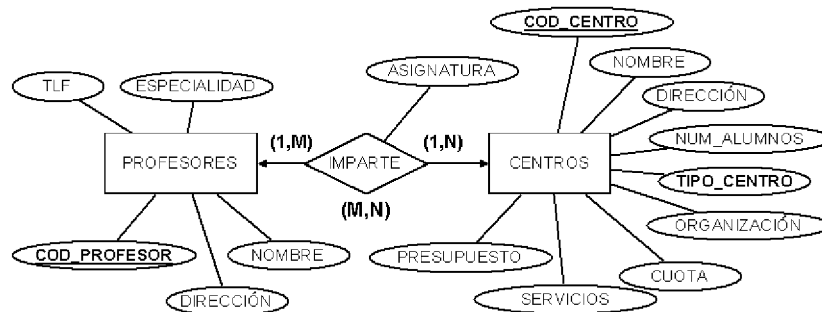


Figura 2.13. *Solución A. Una única entidad.*

El resultado en el modelo relacional contará con las siguientes tablas:

PROFESORES	(**COD_PROFESOR**, DIRECCIÓN, NOMBRE, TLF, ESPECIALIDAD)
CENTROS	(**COD_CENTRO**, NOMBRE, DIRECCIÓN, NUM_ALUMNOS, TIPO_CENTRO, ORGANIZACIÓN, CUOTA, SERVICIOS, PRESUPUESTO)
IMPARTE	(**COD_PROFESOR(FK)**, **COD_CENTRO(FK)**, ASIGNATURA)

B. Eliminación del supertipo, el esquema E-R se muestra en la Figura 2.14:

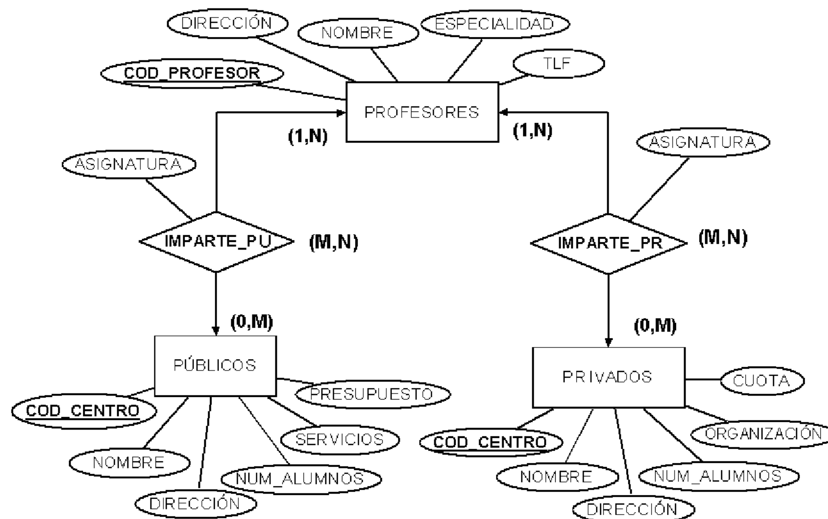


Figura 2.14. *Solución B. Eliminación del supertipo, dos entidades.*

(Continúa)

(Continuación)

El TIPO_CENTRO se puede arrastrar a las subentidades pero no tendría mucho sentido, pues tomaría siempre el mismo valor, es decir, en un centro público sería siempre público y en uno concertado, concertado. Así pues se elimina. El resultado en el modelo relacional contará con las siguientes tablas:

PROFESORES (**COD_PROFESOR**, DIRECCIÓN, NOMBRE, TLF, ESPECIALIDAD)
PÚBLICOS (**COD_CENTRO**, NOMBRE, DIRECCIÓN, NUM_ALUMNOS, SERVICIOS, PRESUPUESTO)
PRIVADOS (**COD_CENTRO**, NOMBRE, DIRECCIÓN, NUM_ALUMNOS, ORGANIZACIÓN, CUOTA)
IMPARTE_PR (**COD_PROFESOR(FK)**, COD_CENTRO(FK), ASIGNATURA)
IMPARTE_PU (**COD_PROFESOR(FK)**, COD_CENTRO(FK), ASIGNATURA)

Si nos fijamos en el esquema relacional resultante vemos que las tablas PUBLICOS y PRIVADOS tiene la misma clave, y no existe ningún condicionante lógico que impida que un centro esté en las dos tablas, algo que va en contra del enunciado del problema. Para controlar esta situación habrá que recurrir a los programas que manejen las tablas o a la creación de triggers asociados a cada una de las tablas.

Del mismo modo, podemos pensar que las tablas IMPARTE_PR e IMPARTE_PU, son iguales y podrían agruparse en una sola, sin embargo, habría problemas con el control de la integridad puesto que habría que definir la clave de esta tabla como clave ajena de las tablas PÚBLICOS y PRIVADOS, y esto no sería correcto.

C. Inserta una relación 1:1 entre el supertipo y cada uno de los subtipos. El esquema E-R se muestra en la Figura 2.15:

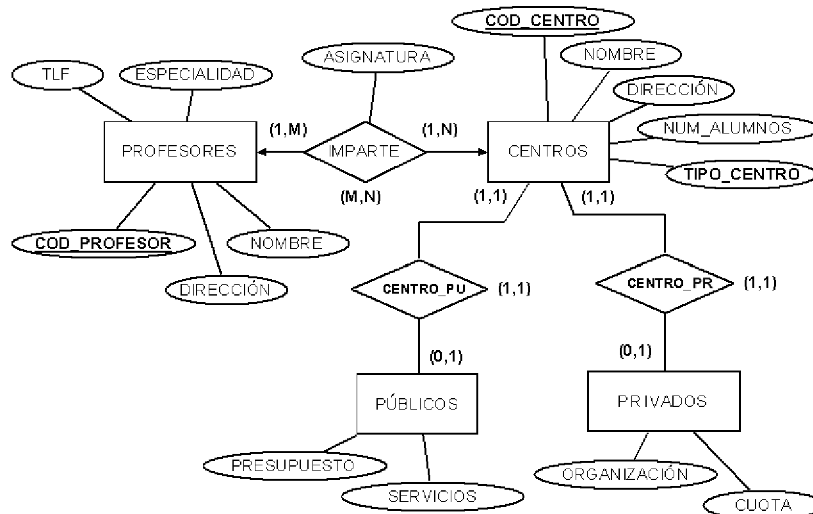


Figura 2.15. *Solución C. Relación 1:1 entre supertipo y subtipos.*

El atributo TIPO_CENTRO se incluye en el supertipo. El resultado en el modelo relacional contará con las siguientes tablas:

PROFESORES (**COD_PROFESOR**, DIRECCION, NOMBRE, TLF, ESPECIALIDAD)
CENTROS (**COD_CENTRO**, NOMBRE, DIRECCION, NUM_ALUMNOS, TIPO_CENTRO)
PÚBLICOS (**COD_CENTRO(FK)**, SERVICIOS, PRESUPUESTO)
PRIVADOS (**COD_CENTRO(FK)**, ORGANIZACIÓN, CUOTA)
IMPARTE (**COD_PROFESOR(FK), COD_CENTRO(FK)**, ASIGNATURA)





- **Relaciones N-arias**

En este tipo de relaciones se agrupan 3 o más entidades, y para pasar al modelo de datos relacional cada entidad se convierte en tabla, así como la relación, que va a contener los atributos propios de ella más las claves de todas las entidades. La clave de la tabla resultante será la concatenación de las claves de las entidades. Hay que tener en cuenta:

– Si la relación es M:M:M, es decir, si todas las entidades participan con cardinalidad máxima M, la clave de la tabla resultante es la unión de las claves de las entidades que relaciona. Esa tabla incluirá los atributos de la relación si los hubiera.

Caso práctico



8 **Suponemos una relación ternaria entre las entidades PROFESORES-CURSOS-ASIGNATURAS, en la que un profesor imparte en varios cursos varias asignaturas, y además las asignaturas son impartidas por varios profesores en varios cursos. El esquema E-R se muestra en la Figura 2.16. Transformarlo al modelo relacional.**

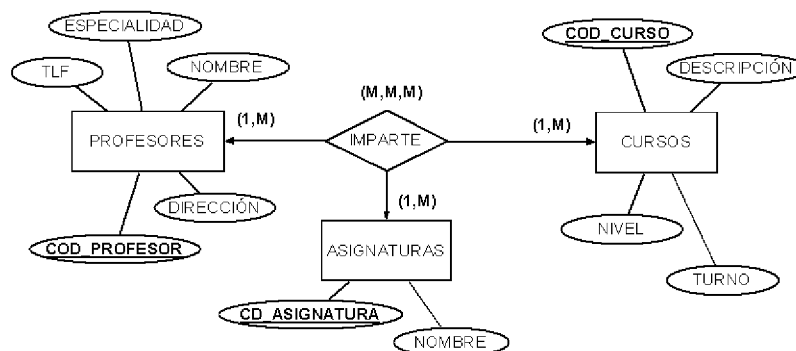


Figura 2.16. *Esquema E-R de PROFESORES-CURSOS-ASIGNATURAS.*

El resultado en el modelo relacional contará con las siguientes tablas:

PROFESORES	(**COD_PROFESOR**, DIRECCIÓN, NOMBRE, TLF, ESPECIALIDAD)
CURSOS	(**COD_CURSO**, DESCRIPCIÓN, NIVEL, TURNO)
ASIGNATURAS	(**CD_ASIGNATURA**, NOMBRE)
IMPARTE	(**COD_PROFESOR(FK), COD_CURSO(FK)**, CD_ASIGNATURA(FK))

– Si la relación es 1:M:M, es decir, una de las entidades participa con cardinalidad máxima 1, la clave de esta entidad no pasa a formar parte de la clave de la tabla resultante, pero forma parte de la relación como un atributo más.



Caso práctico

9 **Suponemos el caso de una tienda de venta de coches, en la que un empleado vende muchos coches a muchos clientes, y los coches son vendidos por un solo empleado. En la venta hay que tener en cuenta la forma de pago y la fecha de venta. El esquema E-R se muestra en la Figura 2.17. Transfórmalo al modelo relacional.**

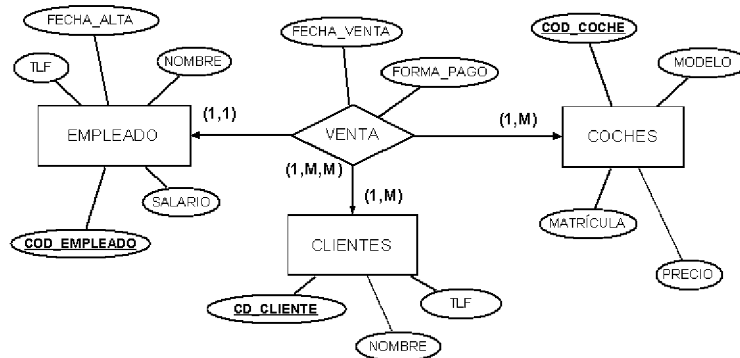


Figura 2.17. *Esquema E-R de EMPLEADO-CLIENTES-COCHES.*

El resultado en el modelo relacional es:

CLIENTES (**CD_CLIENTE**, NOMBRE, TLF)
EMPLEADO (**COD_EMPLEADO**, NOMBRE, TLF, SALARIO, FECHA_ALTA)
COCHES (**COD_COCHE**, MATRICULA, MODELO, PRECIO)
VENTA (**COD_COCHE(FK)**, **CD_CLIENTE(FK)**, COD_EMPLEADO, FORMA_PAGO, FECHA_VENTA)

En la Tabla 2.1 se muestra un resumen de las transformaciones E-R a Relacional:

Modelo Entidad-Relación (Chen)			Modelo relacional
Entidad			Tabla
Relaciones	Binarias	1:N	Propagar la clave: clave ajena (tabla con N), o bien otra tabla.
		1:1	Propagar la clave: clave ajena, o bien otra tabla o bien uniendo las dos entidades en una tabla.
		M:N	Otra tabla nueva, con clave primaria igual a la suma de las claves primarias de las dos tablas.
	Reflexivas	1:N	Clave ajena o bien otra tabla.
		1:1	Clave ajena o bien otra tabla.
		M:N	Otra tabla nueva, relación con clave principal igual a la clave principal de la entidad duplicada.
	Ternarias		Se crea una tabla nueva y a la hora de elegir la clave se tendrá en cuenta: 1. Concatenación de claves primarias de las entidades, con grado diferente a 1 (M,M,M...) 2. Si alguna tiene cardinalida máxima 1, al menos ha de haber (N-1) claves primarias de otras (N-1) entidades, y han de participar en la relación las claves primarias de las entidades con cardinalidad máxima 1.

Tabla 2.1. *Resumen de la conversión del modelo E-R al relacional.*





2.6 Pérdida de semántica en la transformación al modelo relacional

Algunas restricciones son necesarias controlarlas con mecanismos externos al modelo relacional dado que muchos de SGBD no implementan el modelo relacional completo. Por ejemplo, los mecanismos tipo CHECK y ASSERTION no suelen estar disponibles en todos los SGBD lo que implica que hay que recurrir a otros medios para recoger estas restricciones como: el uso de disparadores, procedimientos almacenados o aplicaciones externas.

Algunas de las restricciones de los esquemas E-R que deben contemplarse en la transformación al modelo relacional mediante *checks*, aserciones o disparadores son:

– Cardinalidades mínimas de 1 en relaciones N:M y 1:N (excluyendo aquellas que se controlan con la restricción NOT NULL cuando se realiza una propagación de clave).

– Cardinalidades máximas conocidas en relaciones binarias N:M y 1:N y relaciones ternarias.

– Exclusividad en las generalizaciones.

– Inserciones y borrado en las generalizaciones.

– Restricciones que no figuran en el enunciado original pero que se consideran adecuadas o convenientes (por ejemplo, restricciones que implican operadores de comparación de fechas, etcétera).

2.7 Normalización de esquemas relacionales

La **normalización** es una técnica para diseñar la estructura lógica de los datos de un sistema de información en el modelo relacional, desarrollada por E. F. Codd en 1972. Es una estrategia de diseño de abajo arriba: se parte de los atributos y éstos se van agrupando en relaciones (tablas) según su afinidad. Aquí no se utilizará la normalización como una técnica de diseño de bases de datos, sino como una etapa posterior a la correspondencia entre el esquema conceptual y el esquema lógico, que elimine las dependencias no deseadas entre atributos. Las ventajas de la normalización son las siguientes:

- Evita anomalías en inserciones, modificaciones y borrados.
- Mejora la independencia de datos.
- No establece restricciones artificiales en la estructura de los datos.

Uno de los conceptos fundamentales en la normalización es el de *dependencia funcional*. Una **dependencia funcional** es una relación entre atributos de una misma relación (tabla).

Si X e Y son atributos de la relación R, se dice que Y es funcionalmente dependiente de X (se denota por X → Y) si cada valor de X tiene asociado un solo valor de Y (X e Y pueden constar de uno o varios atributos). A X se le denomina *determinante*, ya que X determina el valor de Y. Se dice que el atributo Y es *completamente dependiente* de X si depende funcionalmente de X y no depende de ningún subconjunto de X.

Esto es lo mismo que decir que si dos tuplas de R tienen el mismo valor para su atributo X, forzosamente han de tener el mismo valor para el atributo Y.

La dependencia funcional es una noción semántica. Si hay o no dependencias funcionales entre atributos no lo determina una serie abstracta de reglas, sino, más bien, los modelos mentales del usuario y las reglas de negocio de la organización o empresa para la que se desarrolla el sistema de información. Cada dependencia funcional es una clase especial de regla de integridad y representa una relación de uno a muchos.

En el proceso de normalización se debe ir comprobando que cada relación (tabla) cumple una serie de reglas que se basan en la clave primaria y las dependencias funcionales. Cada regla que se cumple aumenta el grado de normalización. Si una regla no se cumple, la relación se debe descomponer en varias relaciones que sí la cumplan.

La normalización se lleva a cabo en una serie pasos. Cada paso corresponde a una forma normal que tiene unas propiedades. Conforme se va avanzando en la normalización, las relaciones tienen un formato más estricto (más fuerte) y, por lo tanto, son menos vulnerables a las anomalías de actualización. El modelo relacional sólo requiere un conjunto de relaciones en primera forma normal. Las restantes formas normales son opcionales. Sin embargo, para evitar las anomalías de actualización, es recomendable llegar al menos a la tercera forma normal.

A. Cálculo de dependencias

Las **dependencias** son propiedades inherentes al contenido semántico de los datos formando parte de las restricciones de usuario del modelo relacional. Entre los atributos de una relación pueden existir dependencias de varios tipos:

- **Dependencias funcionales**

  Son de primordial importancia a la hora de encontrar y eliminar la redundancia de los datos almacenados en las tablas de una base de datos relacional, se centran en el estudio de las dependencias que presenta cada atributo de una relación con respecto al resto de atributos de la misma.

  Dada una relación R que contiene los atributos X e Y se dice que Y depende funcionalmente de X (X → Y) si y sólo si en todo momento cada valor de X tiene asociado un solo valor de Y. Esto es lo mismo que decir que si dos tuplas de R tienen el mismo valor para su atributo X forzosamente han de tener el mismo valor para el atributo Y.

  Tipos de dependencias:

  – **Dependencias completa y parcial.** Dado un atributo compuesto X formado por los atributos X1 y X2, se dice que el atributo Y tiene una dependencia funcional completa con respecto a X si: X1 → Y; X2 → Y; Y -/ → X (Y no implica X) pero X → Y.





Por el contrario, dado un atributo compuesto X formado por los atributos X1 y X2, se dice que el atributo Y tiene una dependencia funcional parcial con respecto a X si: X1 – → Y o X2 –/ Y y X → Y

– **Dependencia transitiva.** Dados los atributos X, Y y Z de la relación R en la que existen las siguientes dependencias funcionales X → Y ; Y → Z ; se dice que Z tiene una dependencia transitiva respecto X a través de Y: X → Z. Por ejemplo: Nombre de alumno Dirección; Dirección Ciudad; Nombre de alumno Ciudad.

– **Dependencia multivaluada o de valores múltiples.** Sean X e Y dos descriptores, X multidetermina a Y (X → Y) si para cada valor de X existe un conjunto bien definido de valores posibles en Y, con independencia del resto de los atributos de la relación. Este tipo de dependencias produce redundancia de datos, como se puede apreciar en la Tabla 2.2, en donde las claves 34567-B y 89345-M tienen dos registros para mantener la serie de datos en forma independiente. Esto ocasiona que al realizarse una actualización se requiera de demasiadas operaciones para tal fin.

En la tabla siguiente, el atributo DNI multidetermina a Titulación; ver Tabla 2.2:

DNI	TITULACIÓN
34567-B	Licenciado en Exactas
34567-B	Licenciado en Ciencias Físicas
56667-C	Ingeniero de Telecomunicaciones
89345-M	Licenciado en Historia
89345-M	Licenciado en Filosofía

Tabla 2.2. *Dependencia multivaluada.*

– **Dependencias de reunión o en combinación.** Se dice que una relación satisface la dependencia de reunión (X, Y, … , Z) si y sólo si la relación es igual a la reunión de sus proyecciones según X, Y, … , Z . Donde X, Y, … , Z son subconjuntos del conjunto de atributos de la relación.

- **Reglas de normalización**

Se dice que una relación está en una forma normal si satisface un cierto conjunto específico de restricciones impuestas por la regla de normalización correspondiente. La aplicación de una regla es una operación que toma como entrada una relación y da como resultado dos o más relaciones.

- **Primera forma normal. 1FN**

Se dice que una relación está en 1FN si y sólo si los valores que componen cada atributo de una tupla son atómicos, es decir, cada atributo de la relación toma un único valor del dominio correspondiente, o lo que es lo mismo no existen grupos repetitivos.

La tabla estará en 1FN si tiene un solo valor en la intersección de cada fila con cada columna. Un conjunto de relaciones se encuentra en 1FN si ninguna de ellas tiene grupos repetitivos.

Si una relación no está en 1FN, hay que eliminar de ella los grupos repetitivos. Un grupo repetitivo será el atributo o grupo de atributos que tiene múltiples valores para cada tupla de la relación. Hay dos formas de eliminar los grupos repetitivos:

- Repetir los atributos con un solo valor para cada valor del grupo repetitivo. De este modo, se introducen redundancias ya que se duplican valores, pero estas redundancias se eliminarán después mediante las restantes formas normales.

- La segunda forma de eliminar los grupos repetitivos consiste en poner cada uno de ellos en una relación aparte, heredando la clave primaria de la relación en la que se encontraban.



Caso práctico

10 **La aplicación de esta regla es fácil. Este proceso se realiza casi siempre en el análisis del problema. Por ejemplo, consideramos la tabla ALUMNO, con clave primaria COD_ALUMNO, en la que el atributo TLF puede tomar varios valores: el móvil, el de casa, el del padre, el de la madre, etcétera. Ver Tabla 2.3:**

COD_ALUMNO	NOMBRE	APELLIDO	TLF	DIRECCION
1111	PEPE	GARCÍA	678-900800 91-2233441 91-1231232	C/Las cañas 45
2222	MARÍA	SUÁREZ	91-7008001	C/Mayor 12
3333	JUAN	GIL	91-7562324 660-111222	C/La plaza
4444	FRANCISCO	MONTOYA	678-556443	C/La arboleda

Tabla 2.3. *Tabla ALUMNO que no está en 1FN.*

Esta tabla no está en 1FN, ya que hay dos alumnos que tienen varios teléfonos. Para que esta tabla esté en 1FN se puede:

- Definir como clave primaria de la tabla COD_ALUMNO, y el TLF, con el fin de que cada atributo tome un único valor en la tupla correspondiente. Ver Tabla 2.4.

COD_ALUMNO	TLF	NOMBRE	APELLIDO	DIRECCIÓN
1111	678-900800	PEPE	GARCÍA	C/Las cañas 45
1111	91-2233441	PEPE	GARCÍA	C/Las cañas 45
1111	91-1231232	PEPE	GARCÍA	C/Las cañas 45
2222	91-7008001	MARÍA	SUÁREZ	C/Mayor 12
3333	91-7562324	JUAN	GIL	C/La plaza
3333	660-111222	JUAN	GIL	C/La plaza
4444	678-556443	FRANCISCO	MONTOYA	C/La arboleda

Tabla 2.4. *Tabla ALUMNO, primera transformación a 1FN.*

(Continúa)





(Continuación)

- O también se eliminan los grupos repetitivos (TLF) y se crea una relación (tabla) junto con la clave inicial. Ver Tablas 2.5 y 2.6.

COD_ALUMNO	NOMBRE	APELLIDO	DIRECCION
1111	PEPE	GARCÍA	C/Las cañas 45
2222	MARÍA	SUÁREZ	C/Mayor 12
3333	JUAN	GIL	C/La plaza
4444	FRANCISCO	MONTOYA	C/La arboleda

Tabla 2.5. *Tabla ALUMNO segunda transformación a 1FN.*

La segunda tabla estará formada por el TLF, y el COD_ALUMNO. Los dos formarán la clave. Ver Tabla 2.6:

COD_ALUMNO(FK)	TLF
1111	678-900800
1111	91-2233441
1111	91-1231232
2222	91-7008001

Tabla 2.6. *Tabla ALUM_TLF en 1FN.*

- **Segunda forma normal. 2FN**

Se dice que una relación se encuentra en 2FN si y sólo si satisface la 1FN, y cada atributo de la relación que no está en la clave depende funcionalmente de forma completa de la clave primaria de la relación. La 2FN se aplica a las relaciones que tienen claves primarias compuestas por dos o más atributos.

Si una relación está en 1FN y su clave primaria es simple (tiene un solo atributo), entonces también está en 2FN. Las relaciones que no están en 2FN pueden sufrir anomalías cuando se realizan actualizaciones.

Si la clave primaria está formada por un solo atributo y la relación está en 1FN, ya está en 2FN.

Teorema de la 2FN: sea una relación formada por los atributos A, B, C, D con clave primaria compuesta por los atributos A y B. Si se cumple que (D depende funcionalmente de A): A $\rightarrow$ D, entonces la relación puede descomponerse en dos relaciones R1 y R2 con los atributos respectivos: R1 (A, D) y R2 (A, B, C).

Para pasar una relación en 1FN a 2FN hay que eliminar las dependencias parciales de la clave primaria. Para ello, se eliminan los atributos, que son funcionalmente dependientes, y se ponen en una nueva relación con una copia de su determinante (los atributos de la clave primaria de los que dependen).

Se crearán dos tablas para eliminar las dependencias funcionales, una de ellas tendrá los atributos que dependen funcionalmente de la clave, y la otra los atributos que forman parte de la clave de la que dependen.



Caso práctico

11 **Supongamos que tenemos una relación ALUMNO en la que representamos los datos de los alumnos y las notas en cada una de las asignaturas en que está matriculado. La clave es el número de matrícula COD_ALUMNO y la asignatura ASIGNATURA, ver Tabla 2.7:**

<u>COD_ALUMNO</u>	NOM_ALUM	APE_ALUM	<u>ASIGNATURA</u>	NOTA	CURSO	AULA
1111	PEPE	GARCÍA	LENGUA I	5	1	15
1111	PEPE	GARCÍA	IDIOMA	5	2	16
2222	MARÍA	SUAREZ	IDIOMA	7	2	16
2222	MARÍA	SUAREZ	CIENCIAS	7	2	14
3333	JUAN	GIL	PLÁSTICA	6	1	18
3333	JUAN	GIL	MATEMÁTICAS I	6	1	12
4444	FRANCISCO	MONTOYA	LENGUA II	4	2	11
4444	FRANCISCO	MONTOYA	MATEMÁTICAS I	6	1	12
4444	FRANCISCO	MONTOYA	CIENCIAS	8	1	14

Tabla 2.7. *Relación ALUMNO para transformar a 2FN.*

Es obvio que todos los atributos no dependen de la clave completa (COD_ALUMNO, ASIGNATURA). En primer lugar, hay que ver las dependencias funcionales de cada uno de los atributos con respecto a los atributos de la clave y el resto de atributos:

– NOM_ALUM y APE_ALUM, sólo dependen de COD_ALUMNO.
– Los atributos CURSO y AULA están relacionados con la ASIGNATURA, es decir, existe una dependencia entre ASIGNATURA → CURSO, ASIGNATURA → AULA. Una asignatura pertenece a un curso y se imparte en un aula.
– El atributo NOTA depende funcionalmente de la clave, pues para que haya una nota tiene que haber una asignatura y un alumno.

Vistas las dependencias funcionales llegamos a la siguiente conclusión para que la relación ALUMNO esté en 2FN necesitamos crear tres relaciones: ALUMNO, ASIGNATURAS y NOTAS (ver Tablas 2.8, 2.9 y 2.10):

<u>COD_ALUMNO</u>	NOM_ALUM	APE_ALUM
1111	PEPE	GARCÍA
2222	MARÍA	SUAREZ
3333	JUAN	GIL
4444	FRANCISCO	MONTOYA

Tabla 2.8. *Tabla ALUMNO en 2FN.*

<u>ASIGNATURA</u>	CURSO	AULA
LENGUA I	1	15
IDIOMA	2	16
CIENCIAS	2	14
PLÁSTICA	1	18
MATEMÁTICAS I	1	12
LENGUA II	2	11

Tabla 2.9. *Tabla ASIGNATURAS en 2FN.*

<u>COD_ALUMNO (FK)</u>	<u>ASIGNATURA (FK)</u>	NOTA
1111	LENGUA I	5
1111	IDIOMA	5
2222	IDIOMA	7
2222	CIENCIAS	7
3333	PLÁSTICA	6
3333	MATEMÁTICAS I	6
4444	LENGUA II	4
4444	MATEMÁTICAS I	6
4444	CIENCIAS	8

Tabla 2.10. *Tabla NOTAS en 2FN.*





- **Tercera forma normal. 3FN**

  Una relación está en tercera forma normal si, y sólo si, está en 2FN y, además, cada atributo que no está en la clave primaria no depende transitivamente de la clave primaria. Es decir, los atributos de la relación no dependen unos de otros, dependen únicamente de la clave, esté formada por uno o más atributos. La dependencia X → Z es transitiva si existen las dependencias X → Y, Y → Z, siendo X, Y, atributos o conjuntos de atributos de una misma relación.

  Para pasar una relación de 2FN a 3FN hay que eliminar las dependencias transitivas. Para ello, se eliminan los atributos que dependen transitivamente y se ponen en una nueva relación con una copia de su determinante (el atributo o atributos no clave de los que dependen).

  Teorema de la 3FN: sea una relación formada por los atributos A, B, C con clave primaria formada por el atributo A. Si se cumple que: B → C, entonces la relación puede descomponerse en dos relaciones, R1 y R2, con los atributos respectivos: R1 (A, B) y R2 (B, C).

Caso práctico



12 **Supongamos que tenemos una relación LIBROS en la que representamos los datos de las editoriales de los mismos, ver Tabla 2.11:**

COD_LIBRO	TÍTULO	EDITORIAL	PAÍS
12345	DISEÑO DE BD RELACIONALES	RAMA	ESPAÑA
34562	INSTALACIÓN y MANTENIMIENTO DE EQUIPOS	MCGRAW-HILL	ESPAÑA
72224	FUNDAMENTOS DE PROGRAMACIÓN	SANTILLANA	ESPAÑA
34522	BASE DE DATOS OO	ADDISON	EEUU

Tabla 2.11. *Relación LIBROS para transformar a 3FN.*

Veamos las dependencias con respecto a la clave:

- TÍTULO y EDITORIAL dependen directamente del código del libro.
- El PAÍS, aunque en parte también depende del libro, está más ligado a la EDITORIAL a la que pertenece el libro. Por esta última razón, la relación libros no está en 3FN, la solución se muestra en las Tablas 2.12 y 2.13:

COD_LIBRO	TÍTULO	EDITORIAL (FK)
12345	DISEÑO DE BD RELACIONALES	RAMA
34562	INSTALACIÓN y MANTENIMIENTO DE EQUIPOS	MCGRAW-HILL
72224	FUNDAMENTOS DE PROGRAMACIÓN	SANTILLANA
34522	BASE DE DATOS OO	ADDISON

Tabla 2.12. *Tabla LIBROS en 3FN.*

EDITORIAL	PAÍS
RAMA	ESPAÑA
MCGRAW-HILL	ESPAÑA
SANTILLANA	ESPAÑA
ADDISON	EEUU

Tabla 2.13. *Tabla EDITORIAL en 3FN.*



Actividades propuestas

2 **Dada la tabla 2.14 normalizar hasta la 3FN.**

DNI	NOMBRE	APELLIDOS	DIRECCIÓN	C_POST	POBLACIÓN	PROVINCIA
413245-B	JUAN	RAMOS	C/Las cañas 59 C/Pilón 12	19005 45589	GUADALAJARA CALERUELA	GUADALAJARA TOLEDO
23456-J	PEDRO	PÉREZ	C/Vitoria 3 C/El altozano	28804 10392	ALCALÁ DE HENARES BERROCALEJO	MADRID CÁCERES
34561-B	MARÍA	RODRÍGUEZ	C/Sanz Vázquez 2	19004	GUADALAJARA	GUADALAJARA
222346-J	JUAN	CABELLO	C/El ensanche 3 C/Los abedules 10	28802 10300	ALCALÁ DE HENARES NAVALMORAL DE LA MATA	MADRID CÁCERES

Tabla 2.14. *Transformar a 3FN.*

- **Forma Normal de Boyce - Codd (FNBC)**

Se define determinante en una relación a un atributo del cual depende funcionalmente de manera completa cualquier otro atributo de la relación. Una relación está en la Forma Normal de Boyce - Codd (FNBC) si, y sólo si, todo determinante de ella es una clave candidata.

La 2FN y la 3FN eliminan las dependencias parciales y las dependencias transitivas de la clave primaria. Pero este tipo de dependencias todavía pueden existir sobre otras claves candidatas, si las hubiera. La BCFN es más fuerte que la 3FN, por lo tanto, toda relación en FNBC está en 3FN.

La violación de la FNBC es poco frecuente ya que se da bajo ciertas condiciones que raramente se presentan. Se debe comprobar si una relación viola la FNBC si tiene dos o más claves candidatas compuestas que tienen al menos un atributo en común.

Teorema de Boyce - Codd: sea una relación R formada por los atributos A, B, C, D con claves candidatas compuestas (A, B) y (B, C) tal que: A C, entonces la relación puede descomponerse en cualquiera de las dos siguientes maneras: R1 (A, C) y R2 (B, C, D) o bien, R1 (A, C) y R2 (A, B, D).



Caso práctico

13 **Supongamos que tenemos una relación EMPLEADOS en la que representamos los datos de los empleados de una fábrica, ver Tabla 2.15:**

DNI	NÚM_SEG_SOC	NOMBRE	APELLIDOS	DEPARTAMENTO	PUESTO	SALARIO
413245-B	28-1234566	JUAN	RAMOS	COMPRAS	GERENTE	2.300
23456-J	28-2345686	PEDRO	PÉREZ	NÓMINAS	AUXILIAR	1.200
123123-C	19-458766	MARÍA	GIL	ALMACÉN	CONSERJE	1.530
1234556-B	45-223344	ANTONIO	SANZ	COMPRAS	GESTIÓN	2.200

Tabla 2.15. *Relación EMPLEADOS para transformar a FNBC.*

(Continúa)





(Continuación)

Observando los atributos de esta relación podemos ver que NUM_SEG_SOC y el grupo NOMBRE-APELLIDOS son claves candidatas (determinantes). Esta relación se transforma en dos tablas: una contendrá la clave junto con las claves candidatas (EMPLEADOS) y la otra la clave con el resto de campos (EMPLE_TRABAJO), ver Tablas 2.16 y 2.17:

DNI	NUM_SEG_SOC	NOMBRE	APELLIDOS
413245-B	28-1234566	JUAN	RAMOS
23456-J	28-2345686	PEDRO	PÉREZ
123123-C	19-458766	MARÍA	GIL
1234556-B	45-223344	ANTONIO	SANZ

Tabla 2.16. *Relación EMPLEADOS e FNBC.*

DNI (FK)	DEPARTAMENTO	PUESTO	SALARIO
413245-B	COMPRAS	GERENTE	2.300
23456-J	NÓMINAS	AUXILIAR	1.200
123123-C	ALMACÉN	CONSERJE	1.530
1234556-B	COMPRAS	GESTIÓN	2.200

Tabla 2.17. *Relación EMPLE_TRABAJO en FNBC.*

- **Cuarta Forma Normal (4FN)**

  Una relación se encuentra en 4FN si, y sólo si, está en FNBC y no existen dependencias multivaluadas.

  Teorema de Fagin: Dada la relación formada por los atributos X, Y, Z con las siguientes dependencias multivaluadas: $X \rightarrow\rightarrow Y$ y $X \rightarrow\rightarrow Z$, entonces la relación puede descomponerse en dos relaciones: R1 (X, Y) y R2 (X, Z).

Caso práctico



14 **Consideramos la siguiente relación: GEOMETRÍA, con tres atributos con dependencias multivaluadas, los tres atributos forman la clave, ver Tabla 2.18:**

FIGURA	COLOR	TAMAÑO
ESFERA	ROJO	GRANDE
ESFERA	VERDE	GRANDE
CUBO	BLANCO	GRANDE
CUBO	AZUL	GRANDE
PIRÁMIDE	BLANCO	MEDIANO
PIRÁMIDE	BLANCO	GRANDE
PIRÁMIDE	ROJO	GRANDE

Tabla 2.18. *Relación GEOMETRÍA con dependencias multivaluadas.*

(Continúa)

(Continuación)

En la Tabla 2.18, FIGURA determina valores múltiples de COLOR y TAMAÑO, pero COLOR y TAMAÑO son independientes entre sí. Así pues, las dependencias son FIGURA →→ COLOR y FIGURA →→ TAMAÑO. Observamos que se repiten ESFERA-GRANDE, o PIRÁMIDE-BLANCO, o PIRÁMIDE-GRANDE. Para eliminar la redundancia de los datos se deben eliminar las dependencias de valores múltiples. Esto se logra construyendo dos tablas, donde cada una almacena datos para solamente uno de los atributos de valores múltiples.

Estas repeticiones hacen que la tabla no esté en 4FN. Para pasarla a 4FN, creamos las Tablas 2.19 y 2.20:

FIGURA	COLOR
ESFERA	ROJO
ESFERA	VERDE
CUBO	BLANCO
CUBO	AZUL
PIRÁMIDE	BLANCO
PIRÁMIDE	ROJO

Tabla 2.19. *Tabla COLOR en 4FN.*

FIGURA	TAMAÑO
ESFERA	GRANDE
CUBO	GRANDE
CUBO	MEDIANO
PIRÁMIDE	GRANDE

Tabla 2.20. *Tabla TAMAÑOS en 4FN.*



Actividades propuestas

3 **Normaliza la Tabla 2.21. Suponemos que los estudiantes de una academia pueden inscribirse en varias especialidades y en varios cursos. Por ejemplo, el número de matrícula 12345 tiene especialidades en Sistemas y Telemática y toma los cursos de Natación y Danza.**

NUMATRICULA	ESPECIALIDAD	CURSO
12345	SISTEMAS	NATACIÓN
12345	TELEMÁTICA	DANZA
34567	SISTEMAS	PIANO
34567	DESARROLLO	NATACIÓN
67891	TELEMÁTICA	AERÓBIC

Tabla 2.21. *Tabla ACADEMIA.*

- **Quinta Forma Normal (5FN)**

  Una relación se encuentra en 5FN si, y sólo si, toda dependencia de reunión en la relación es una consecuencia de las claves candidatas. Esto es, la relación estará en 5FN si está en 4FN y no existen restricciones impuestas por el creador de la base de datos. La 5FN se refiere a dependencias que son extrañas. Tiene que ver con tablas que pueden dividirse en subtablas, pero que no pueden reconstruirse. Su valor práctico es ambiguo ya que conduce a una gran división de tablas.





2.8 Dinámica del modelo relacional: álgebra relacional

Hasta ahora, hemos visto la estructura y restricciones del modelo relacional. Es lo que se llama la **componente estática o estática del modelo relacional.** Todo modelo de datos lleva asociado a su parte estática una dinámica que permite la transformación de un estado origen a un estado final de la base de datos. Esta transformación se realiza aplicando un conjunto de operadores mediante los cuales se podrán realizar operaciones de inserción, borrado, modificación y consulta de tuplas.

La dinámica del modelo relacional actúa sobre conjuntos de tuplas y se expresa mediante lenguajes de manipulación relacionales que asocian una sintaxis a las operaciones. En lo que al álgebra se refiere la dinámica la constituye una serie de operadores que aplicados a las relaciones dan como resultado nuevas relaciones.

Operaciones básicas sobre tablas

En este apartado, se muestran diferentes tipos de operaciones de consulta que se pueden realizar con las tablas. Estas operaciones están basadas en el álgebra relacional. Los operandos de cada operación los constituyen una o varias tablas, y el resultado es otra tabla. Esta tabla resultante puede ser, a su vez, operando en otra operación.

Existen dos tipos de operaciones sobre tablas: *básicas* y *derivadas*. Las operaciones básicas se dividen, de nuevo, en *unarias* y *binarias*. En las operaciones unarias se utiliza una tabla de entrada para obtener el resultado; en las binarias se emplean dos tablas.

Las **operaciones derivadas** son binarias que necesitan las operaciones básicas, y son la *intersección*, el *cociente* y la *combinación* o *join*. La Figura 2.18 resume los tipos de operaciones básicas y derivadas.

OPERACIONES BÁSICAS	Operaciones unarias	Selección
		Proyección
	Operaciones binarias	Unión
		Diferencia
		Producto cartesiano
OPERACIONES DERIVADAS	Intersección	
	Cociente	
	Combinación	

Figura 2.18. *Operaciones básicas y derivadas sobre tablas.*

- **Operaciones básicas unarias**
  - **Selección.** Esta operación obtiene un subconjunto de filas de una tabla con todas sus columnas. Se pueden seleccionar determinadas filas incluyendo en la operación una condición. En ésta se pueden utilizar los operadores booleanos: Y (AND), O (OR) , NO (NOT) para expresar varios criterios. Se representa de la siguiente manera: $\sigma_{\text{condición}}$(Tabla).



Caso práctico

15 **A partir de la tabla EMPLEADOS mostrada en la Figura 2.19, seleccionamos aquellas filas cuyo departamento es el 20.**

Nº. EMPLE	APELLIDO	SALARIO	COMISIÓN	Nº. DEPART	JEFE
7369	SÁNCHEZ	1.040		20	7902
7499	ARROYO	2.080	390	30	7698
7521	SALA	1.625	650	30	7698
7566	JIMÉNEZ	3.867		20	7839
7654	MARTÍN	1.625	1820	30	7698
7698	NEGRO	3.705		30	7839
7782	CEREZO	3.185		10	7839
7788	GIL	3.900		20	7566
7839	REY	6.500		10	
7876	ALONSO	1.430		20	7788

Figura 2.19. *Tabla EMPLEADOS con 10 filas.*

Representamos la selección de la siguiente manera: $\sigma_{\text{NºDEPART=20}}$(EMPLEADOS). La tabla resultante se muestra en la Figura 2.20.

Nº. EMPLE	APELLIDO	SALARIO	COMISIÓN	Nº. DEPART	JEFE
7369	SÁNCHEZ	1.040		20	7902
7566	JIMÉNEZ	3.867		20	7839
7788	GIL	3.900		20	7566
7876	ALONSO	1.430		20	7788

Figura 2.20. *Empleados del departamento 20.*



Actividades propuestas

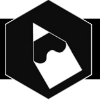
4 **A partir de la tabla EMPLEADOS mostrada en la Figura 2.19, selecciona aquellas filas cuyo JEFE se corresponde con el número 7839 y el departamento es el 30.**

– **Proyección.** Esta operación da como resultado una nueva tabla a partir de otra con el subconjunto de columnas indicado. Las filas duplicadas sólo aparecerán una vez. La proyección se representa de la siguiente manera: $\pi_{col1,col2,..}$(Tabla).





Caso práctico



16 **Proyectamos la tabla EMPLEADOS según las columnas APELLIDO y SALARIO. Se representa así: $\pi_{APELLIDO,SALARIO}(EMPLEADOS)$. La tabla resultante es la de la Figura 2.21.**

APELLIDO	SALARIO
SÁNCHEZ	1.040
ARROYO	2.080
SALA	1.625
JIMÉNEZ	3.867
MARTÍN	1.625
NEGRO	3.705
CEREZO	3.185
GIL	3.900
REY	6.500
ALONSO	1.430

Figura 2.21. *Proyección de la tabla EMPLEADOS según las columnas APELLIDO y SALARIO.*

Obtenemos las columnas APELLIDO y SALARIO de aquellas filas de la tabla EMPLEADOS cuyo departamento es 20. Se representa así: $\pi_{APELLIDO,SALARIO}(\sigma_{NºDEPART=20}(EMPLEADOS))$. La tabla resultante es la de la Figura 2.22.

APELLIDO	SALARIO
SÁNCHEZ	1.040
JIMÉNEZ	3.867
GIL	3.900
ALONSO	1.430

Figura 2.22. *APELLIDO y SALARIO de los empleados del departamento 20.*

Actividades propuestas



5 **A partir de la tabla EMPLEADOS mostrada en la Figura 2.19, obtén el APELLIDO y SALARIO de aquellos empleados cuyo JEFE se corresponde con el número 7839 y el departamento sea el 30.**

- **Operaciones básicas binarias**
  - **Unión.** Dos tablas se pueden unir si tienen el mismo número de columnas y dominios compatibles. El resultado de la unión es otra tabla con las filas de ambas tablas. Las filas repetidas aparecen una sola vez. La unión de dos tablas se representa de la siguiente manera: `Tabla ∪ Tabla2.`



Caso práctico

17 **La unión de las tablas EMPLE1 y EMPLE2 se muestra en la Figura 2.23.**

EMPLE1

N.º emple	Nombre
1001	Rosa
1005	Fernando

EMPLE2

N.º emple	Nombre
2001	Pilar
2010	Octavio
1005	Fernando

EMPLE1 ∪ EMPLE2

N.º emple	Nombre
1001	Rosa
1005	Fernando
2001	Pilar Octavio
2010	

Figura 2.23. *Unión de las tablas EMPLE1 y EMPLE2.*

– **Diferencia.** La diferencia entre dos tablas sólo es posible si tienen el mismo número de columnas y dominios compatibles. El resultado es otra tabla con las filas pertenecientes a una de las tablas y no pertenecientes a la otra tabla. Se representa así: `Tabla – Tabla2`.



Caso práctico

18 **La Figura 2.24 refleja la diferencia de las tablas EMPLE1 y EMPLE2.**

EMPLE2 – EMPLE1

Nº. emple	Nombre
1001	Rosa

EMPLE1 – EMPLE2

Nº. emple	Nombre
2001	Pilar
2010	Octavio

Figura 2.24. *Diferencia de las tablas EMPLE1 y EMPLE2.*



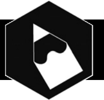
Actividades propuestas

6 **A partir de las tabla AUTORES (Nombre, Editorial) y EDITORES (Nombre, Editorial) inventa un contenido para sus filas y columnas, y representa las operaciones de la unión y de la diferencia.**




– **Producto cartesiano.** Se puede realizar entre dos tablas que tengan diferente número de columnas. El resultado es otra tabla que contendrá la suma de columnas de ambas tablas y el conjunto formado por todas las filas de ambas tablas. No pueden existir columnas con el mismo nombre. El producto se representa de este modo: `Tabla1 X Tabla2`.

Caso práctico



19 **En la Figura 2.25 se muestra el producto cartesiano de las tablas VENTAS y ARTÍCULOS.**

VENTAS

Codi	Fecha	Cantidad
5100	18/11/05	100
5200	19/11/05	120
5100	19/11/05	45

ARTICULOS

Código	Denominación	Existencias	PVP
5100	Patatas	500	3,2
5200	Cebollas	250	4,1

VENTAS X ARTICULOS

Codi	Fecha	Cantidad	Código	Denominación	Existencias	PVP
5100	18/11/05	100	5100	Patatas	500	3,2
5100	18/11/05	100	5200	Cebollas	250	4,1
5200	19/11/05	120	5100	Patatas	500	3,2
5200	19/11/05	120	5200	Cebollas	250	4,1
5100	19/11/05	45	5100	Cebollas	500	3,2
5100	19/11/05	45	5200		250	4,1

Figura 2.25. *Producto cartesiano de las tablas VENTAS y ARTICULOS.*

Actividades propuestas



7 **A partir de las tablas EMPLEADOS (NºEmple, Apellido, Salario, Comision) y DEPARTAMENTOS (NºDepart, Nombre, Localidad) inventa un contenido para sus filas y columnas y representa la operación del producto cartesiano.**

- **Operaciones derivadas**

– **Intersección.** Es una operación derivada de la diferencia. La intersección de dos tablas es otra tabla formada por las filas que aparecen en ambas tablas y las columnas de una de las tablas. Al igual que en la diferencia, las tablas han de tener el mismo número de columnas y dominios compatibles. La operación de la diferencia es la siguiente: `Tabla1-(Tabla1-Tabla2)`. La intersección se representa de la siguiente manera: `Tabla1∩Tabla2`. La Figura 2.26 muestra la intersección de las tablas EMPLE1 y EMPLE2.

EMPLE1∩EMPLE2

Nº. emple	Nombre
1005	Fernando

Figura 2.26. *Intersección de las tablas EMPLE1 y EMPLE2.*

– **Cociente.** El cociente se realiza entre dos tablas, Tabla1 y Tabla2, con diferentes columnas que cumplen las siguientes condiciones: Tabla1 debe tener columnas de Tabla2 y el número de columnas ha de ser mayor que el de Tabla2. Tabla2 debe tener, al menos, una fila. Se puede expresar en función de la proyección, del producto cartesiano y de la diferencia de la siguiente forma:

`Tabla1:Tabla2 = πc(Tabla1) - πc(Tabla2 X πc(Tabla1) - Tabla1)`

Donde C representa el conjunto de atributos de la Tabla1 menos los atributos de la Tabla2.



Caso práctico

20 **En la Figura 2.27 se refleja el cociente de las tablas COCHES y COLORES. La expresión es la siguiente:**

COCHES

MODELO	COLOR
Seat	Rojo
Renault	Verde
Ford	Azul
Ford	Negro
Seat	Azul

COLORES

COLOR
Rojo
Azul

COCHES : COLORES

MODELO
Seat

COCHES:COLORES= π_{MODELO}(COCHES)- π_{MODELO}(COLORES X π_{MODELO}(COCHES)- COCHES)

MODELO
Seat
Renault
Ford
Ford
Seat

MODELO	COLOR
Seat	Rojo
Seat	Azul
Renault	Rojo
Renault	Azul
Ford	Rojo
Ford	Azul

MODELO	COLOR
Renault	Rojo
Renault	Azul
Ford	Rojo

MODELO
Renault
Ford

*(COCHES X π_{MODELO}(COLORES)- COCHES)
se eliminan filas duplicadas

Figura 2.27. *División de las tablas COCHES y COLORES.*

Este operador es útil para simplificar consultas del tipo de la presentada en la figura anterior donde se desea obtener todos los modelos de coches para los que hay todos los colores, contenidos en la tabla COLORES.




Actividades propuestas



8 **A partir de las tablas AUTORES(Nombre, Provincia, Editorial) y EDITORIALES(Editorial) inventa un contenido para sus filas y columnas y representar la operación de División (AUTORES:EDITORIALES). ¿Qué representa el resultado final?**

– **Combinación o *join*.** Con esta operación se obtiene el producto cartesiano de dos tablas cuyas filas cumplan una condición determinada. La combinación de dos tablas con respecto a una cierta condición de combinación es otra tabla constituida por pares de filas de ambas tablas concatenadas tales que en cada par las correspondientes filas satisfacen la condición. Se representa del siguiente modo: `(Tabla1*Tabla2)`$_{condición}$.

Caso práctico



21 **La Figura 2.28 muestra la combinación de las tablas VENTAS y ARTÍCULOS vistas en el Caso práctico 19. La combinación de ambas tablas da lugar a una nueva tabla en la que las filas de cada venta contienen los datos del artículo correspondiente. El criterio de combinación es la igualdad en el código de artículo de ambas tablas. Los códigos de artículo están representados por las columnas CODI y CÓDIGO.**

(CUENTAS X ARTÍCULOS) $_{\text{CODI=CÓDIGO}}$

Codi	Fecha	Cantidad	Denominación	Existencias	PVP
5100	18/11/05	100	Patatas	500	3,2
5200	19/11/05	120	Cebollas	25	4,1
5100	19/11/05	45	Patatas	500	3,2

Figura 2.28. *Combinación de las tablas VENTAS y ARTÍCULOS.*

Actividades propuestas



9 **A partir de las tablas EMPLEADOS (Nº.Emple, Apellido, Salario, Comisión, Nº.Depart, Jefe) y DEPARTAMENTOS(Nº.Departamento, NombreDepartamento, Localidad) inventa un contenido para sus filas y columnas y representa la operación de Combinación. Obtén después una expresión donde se obtengan sólo las columnas: Nº.Emple, Apellido, Salario y NombreDepartamento.**

Conceptos básicos



La Figura 2.29 muestra un esquema básico de la estructura del modelo relacional.

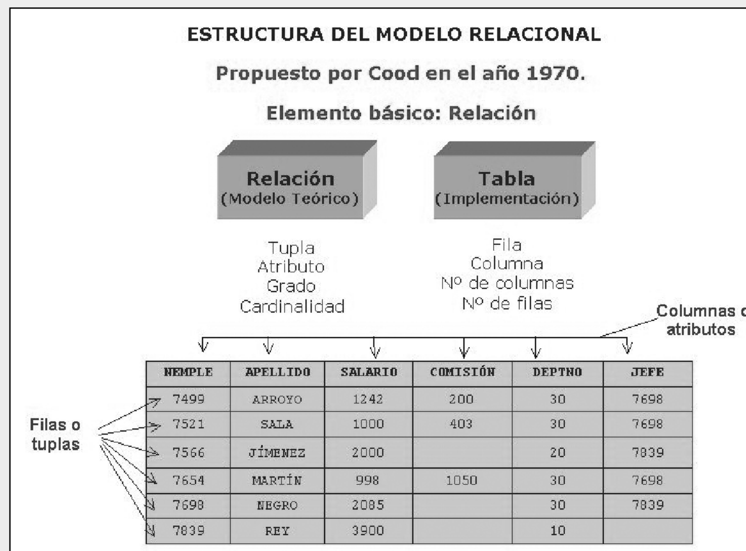


Figura 2.29. *Estructura del Modelo relacional.*

En la Figura 2.30 se muestra un esquema básico de las operaciones básicas sobre tablas basadas en el álgebra relacional.

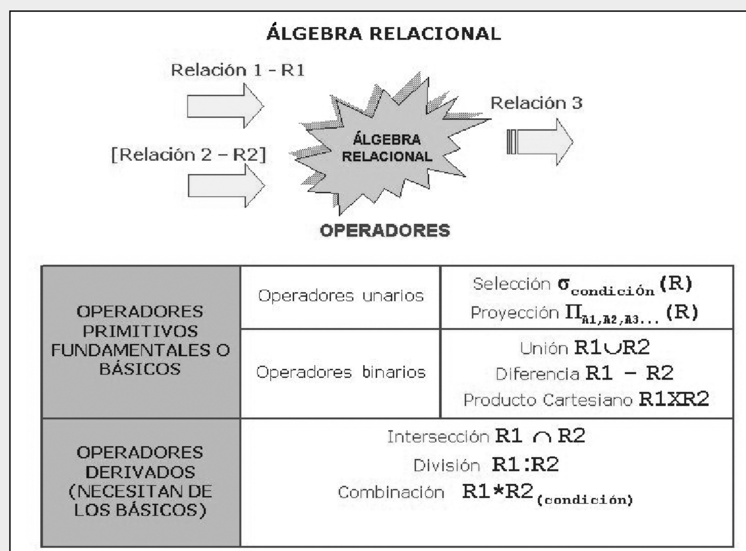


Figura 2.30. *Operaciones básicas en algebra relacional.*





Actividades complementarias

1 **Se desea informatizar la gestión de los proyectos del departamento de química de una universidad siguiendo las siguientes especificaciones:**

- Al departamento llegan una serie de clientes que quieren realizar proyectos. Generalmente los clientes son empresas que realizan contratos con el grupo de investigación del departamento. Un cliente puede realizar varios proyectos.
- Un proyecto es de un cliente. Cada proyecto tiene asignada una cuantía de dinero que se utilizará para pagar los gastos del proyecto. De esta cuantía se saca el dinero para realizar los pagos a los colaboradores. También nos interesa saber de los proyectos el nombre, la fecha de comienzo, la de fin, entre otros.
- De cada proyecto se realizan muchos pagos para pagar a los colaboradores.
- De los pagos nos interesa saber el concepto, la cantidad, el IVA aplicado y la fecha del pago.
- Existen varios tipos de pagos (por ejemplo, Nómina, Representación, Material, etcétera). Un pago es de un tipo de pago, y a un tipo de pago pueden pertenecer muchos pagos.
- Existen una serie de colaboradores que son personas o entidades que van a recibir el dinero de los pagos en concepto de una tarea realizada o la compra de material. Un pago sólo puede ser para un colaborador. Éste a su vez puede recibir muchos pagos.
- De los colaboradores nos interesa saber: Nombre, NIF, Domicilio, Teléfono, Retención, Banco, Nº. Cuenta.

Realiza el diagrama E-R que cumpla las especificaciones y pásalo al modelo de datos relacional.

2 **Dada la tabla 2.22, transformarla a 3FN.**

COD_EMPLE	NOMBRE	COD_DEP	NOMBRE_DEP	AÑOS_DEP
1	Juan	6	Contabilidad	6
2	Pedro	3	Sistemas	3
2	Pedro	6	Contabilidad	5
3	Sonia	2	I+D	1
4	Verónica	3	Sistemas	10
4	Verónica	6	Contabilidad	2

Tabla 2.22. *Tabla ejercicio 2.*

3 **A partir de las siguientes tablas:**

AGENDA (Nombre, Edad, CódigoProvincia, Telef)
PROVINCIAS (Código, Nombreprov)

- Escribe la columna o conjunto de columnas que pueden ser claves primarias y ajenas.
- Escribe un enunciado para las siguientes expresiones:
  a) $\sigma_{Edad>37}$(AGENDA)
  b) $\pi_{Nombre,\ Edad}$(AGENDA)
  c) (AGENDA*PROVINCIAS) $_{CodigoProvincia\ =\ Codigo}$
  d) $\pi_{Nombre,\ CodigoProvincia,\ Telef,\ NombreProv}$((AGENDA*PROVINCIAS)$_{CodigoProvincia\ =\ Codigo}$)
  e) $\pi_{Nombre,CodigoProvincia,Telef,NombreProv}$($\sigma_{Edad>37}$ ((AGENDA* PROVINCIAS)$_{CodigoProvincia\ =\ Codigo}$))

4 **A partir de las siguientes tablas:**

ALUMNOS(DNI, Nombre, Dirección, Telef, Curso)
ASIGNATURAS(Códigoasig, Nombreasig)
NOTAS(DNINo, Códigoasignatura, Nota)

a) Define las claves primarias y ajenas.
b) Obtén expresiones relacionales para:
   - los alumnos de primer curso (Curso=1).
   - los alumnos de primer curso, sólo las columnas DNI y Nombre.
   - el Nombre de los alumnos, el Código de asignatura y la Nota de los alumnos de primer curso.
   - los DNI de los alumnos cuyo nombre de asignatura es INFORMÁTICA.

3 Introducción a SQL

En esta unidad aprenderás a:

1. **Construir sentencias SQL.**
2. **Utilizar el lenguaje SQL para realizar consultas y subconsultas a la base de datos.**
3. **Usar las diferentes cláusulas de consulta con la sentencia SELECT.**
4. **Realizar consultas y subconsultas combinando varias tablas de la base de datos.**





3.1 Introducción

El **lenguaje SQL** (*Structured Query Language*) es una herramienta para organizar, gestionar y recuperar datos almacenados en una base de datos relacional, por tanto, permite la comunicación con el sistema de gestión de la base de datos. Es tan conocido en bases de datos relacionales que muchos lenguajes de programación incorporan sentencias SQL como parte de su repertorio; tal es el caso de Visual Basic. Entre las principales características del SQL podemos destacar las siguientes:

- Es un lenguaje para todo tipo de usuarios (administradores, desarrolladores y usuarios normales).
- El usuario que emplea SQL especifica qué quiere, no dónde ni cómo.
- Permite hacer cualquier consulta de datos.
- Es posible manejarlo para consultas, actualizaciones, definición de datos y control.

Se puede usar de forma interactiva (el usuario escribe las órdenes desde el teclado de un terminal y al instante obtiene los resultados en la pantalla) y de forma embebida (mezclando las instrucciones SQL con las instrucciones propias del lenguaje, tal es el caso del lenguaje PL/SQL).

La Figura 3.1 muestra cómo funciona SQL en una arquitectura cliente/servidor. Por un lado, se dispone de una máquina servidora con una base de datos que contiene datos importantes para el negocio. Por otro lado, tenemos una máquina cliente con un usuario que está ejecutando una aplicación que necesita acceder a los datos allí almacenados. La aplicación realiza una petición al sistema gestor de base de datos, este la procesa y envía los datos a la aplicación que los solicitó.

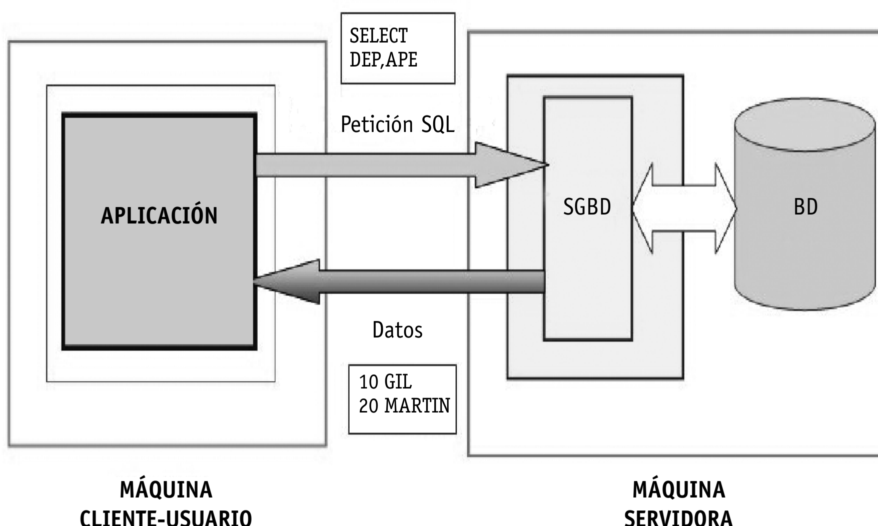


Figura 3.1. *Cómo funciona SQL.*

3.2 Historia y estandarización

El lenguaje SQL fue desarrollado sobre un prototipo de gestor de bases de datos relacionales denominado SYSTEM R y diseñado por IBM a mediados de los años setenta. Incluía lenguajes de consultas, entre ellos **SEQUEL** (*Structured English Query Language*). Más tarde se renombró como SQL.

En 1979 Oracle Corporation presentó la primera implementación comercial de SQL, que estuvo disponible antes que otros productos de IBM. Por su parte, IBM desarrolló productos herederos del prototipo SYSTEM R, como DB2 y SQL/DS.

El instituto **ANSI** (*American National Standard Institute*) adoptó el lenguaje SQL como estándar para la gestión de bases de datos relacionales en octubre de 1986. En 1987 lo adopta **ISO** (*International Standardization Organization*).

En 1989 el estándar **ANSI/ISO,** revisado y ampliado, se llamó **SQL-89** o estándar **SQL1.** Tres años más tarde se aprueba el estándar **ANSI SQL2** o **SQL-92.** En 1999 se aprueba el estándar **SQL:1999** que introduce mejoras con respecto al anterior. **SQL:2003** es el actual estándar. Revisa todos los apartados de SQL:1999 y añade uno nuevo, el apartado 14: **SQL/XML.**

La Figura 3.2 muestra un esquema con la evolución de los estándares y las novedades que va incorporando cada uno con respecto al anterior.

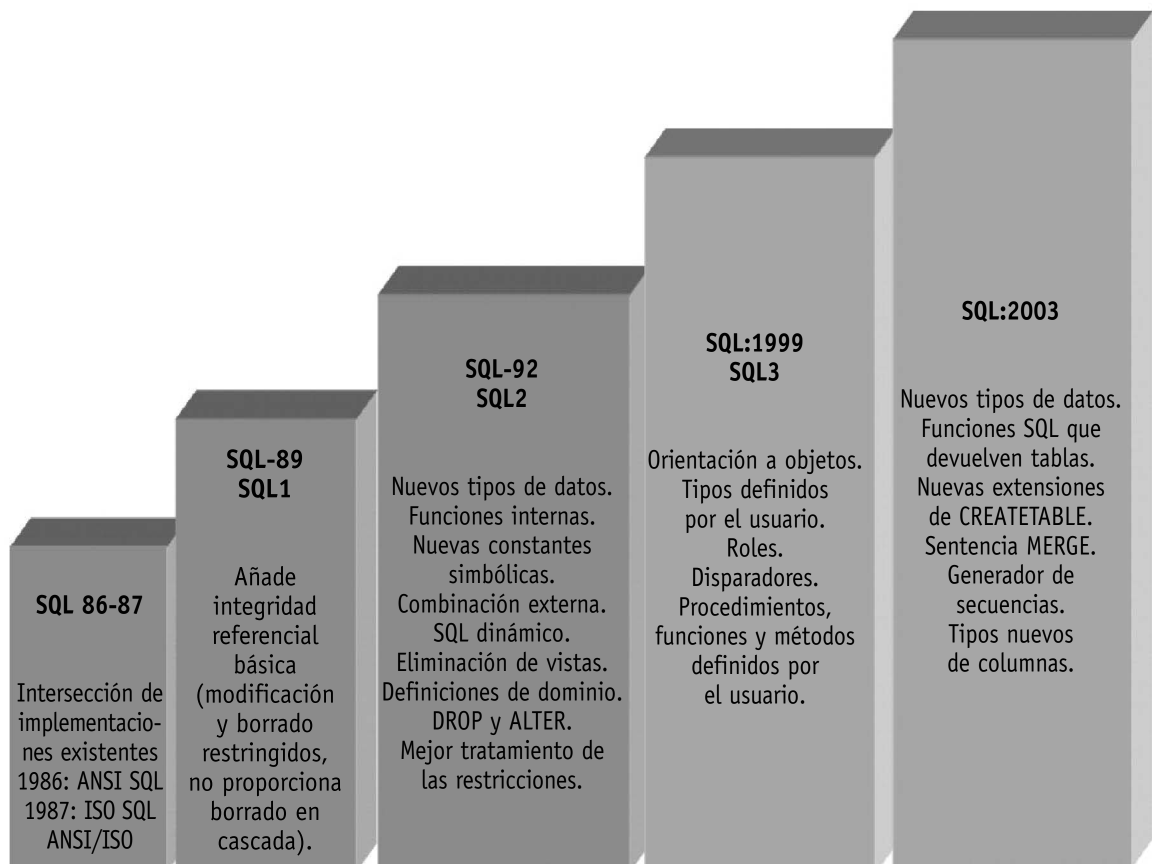


Figura 3.2.
Estándares SQL.





3.3 Tipos de sentencias SQL

El lenguaje SQL proporciona un gran repertorio de sentencias que se utilizan en variadas tareas, como consultar datos de la base de datos, crear, actualizar y eliminar objetos de la base de datos, crear, actualizar y eliminar datos de los objetos, controlar el acceso a la base de datos y a los objetos. Dependiendo de las tareas, podemos clasificar las sentencias SQL en varios tipos. En la Tabla 3.1 se han expuesto las sentencias más importantes y las usadas con mayor frecuencia.

	SENTENCIA	DESCRIPCIÓN
DML	***Manipulación de datos***	
	SELECT	Recupera datos de la base de datos.
	INSERT	Añade nuevas filas de datos a la base de datos.
	DELETE	Suprime filas de datos de la base de datos.
	UPDATE	Modifica datos existentes en la base de datos.
DDL	***Definición de datos***	
	CREATE TABLE	Añade una nueva tabla a la base de datos.
	DROP TABLE*	Suprime una tabla de la base de datos.
	ALTER TABLE*	Modifica la estructura de una tabla existente.
	CREATE VIEW*	Añade una nueva vista a la base de datos.
	DROP VIEW*	Suprime una vista de la base de datos.
	CREATE INDEX*	Construye un índice para una columna.
	DROP INDEX*	Suprime el índice para una columna.
	CREATE SYNONYM*	Define un alias para un nombre de tabla.
	DROP SYNONYM*	Suprime un alias para un nombre de tabla.
DCL	***Control de acceso***	
	GRANT	Concede privilegios de acceso a usuarios.
	REVOKE	Suprime privilegios de acceso a usuarios.
	Control de transacciones	
	COMMIT	Finaliza la transacción actual.
	ROLLBACK	Aborta la transacción actual.
	SQL Programático	
	DECLARE	Define un cursor para una consulta.
	OPEN	Abre un cursor para recuperar resultados de consulta.
	FETCH	Recupera una fila de resultados de consulta.
	CLOSE	Cierra un cursor.

* No existían en el estándar SQL1

Tabla 3.1. *Tipos de sentencias SQL.*

A. Componentes sintácticos de una sentencia

Casi todas las sentencias SQL tienen una forma básica. Véase la Figura 3.3. Todas comienzan con un verbo, que es una palabra clave que describe lo que hace la sentencia (por ejemplo SELECT, INSERT, UPDATE, CREATE). A continuación, le siguen una o más cláusulas que especifican los datos con los que opera la sentencia. Estas también comienzan con una palabra clave como WHERE o FROM. Algunas son opcionales y otras obligatorias. Muchas contienen nombres de tablas o de columnas, palabras reservadas, constantes o expresiones adicionales.

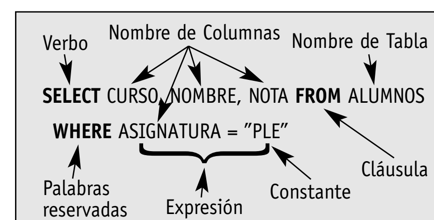


Figura 3.3. *Componentes de una sentencia SQL.*

B. Cómo procesa el SGBD una sentencia

Para procesar una sentencia SQL el sistema gestor de base de datos recorre una serie de pasos:

1. *Analiza la sentencia*. Comprueba que está sintácticamente bien escrita.
2. *Valida la sentencia*. Comprueba la sentencia semánticamente. Es decir, comprueba si las tablas y las columnas referenciadas en la sentencia existen, si el usuario tiene privilegios para realizar esa operación sobre la tabla, etcétera.
3. *Optimiza la sentencia*. El sistema gestor de base de datos explora la forma de llevar a cabo la ejecución de la sentencia.
4. *Genera un plan de aplicación para la sentencia*. Se genera código ejecutable.
5. *Ejecuta el plan de aplicación*.

El análisis de la sentencia no requiere acceso a la base de datos y, por tanto, suele realizarse rápidamente. Sin embargo, la optimización es un proceso muy intensivo de CPU y precisa acceder a la base de datos.

3.4 Tipos de datos

Cuando se crea una tabla instrucción CREATE TABLE (ver Unidad 7) se debe especificar el tipo de dato para cada una de sus columnas, estos tipos de datos definen el dominio de valores que cada columna puede contener. La Tabla 3.2 muestra los distintos tipos de datos de Oracle SQL. Los códigos de los tipos de datos se emplean internamente por Oracle. El código del tipo de dato de una columna es devuelto cuando se usa la función DUMP (ver Unidad 4).

CÓDIGO	Tipo de dato	Características
1	VARCHAR2(tamaño)	Almacena cadenas de caracteres de longitud variable, la longitud se expresa en *tamaño*. La longitud máxima es de 4.000 caracteres (bytes), y la mínima es 1. Ejemplo: DIRECCION VARCHAR2(40)
96	CHAR(tamaño)	Almacena caracteres con una de longitud fija especificada en *tamaño*. La longitud máxima es de 2.000 bytes. El mínimo tamaño y tamaño por defecto es 1 byte. Ejemplo: CODIGO CHAR(6)
2	NUMBER(precisión, escala)	Este tipo almacena datos numéricos, tanto enteros como decimales, con o sin signo. *Precisión* representa el número total de dígitos que va a tener el dato que se define; el rango va de 1 a 38. *Escala* representa el número de dígitos a la derecha del punto decimal. El rango va de -84 a 127. Si se especifica una escala negativa, el número es redondeado tantos dígitos a la izquierda del punto decimal como se indicó en la escala. Si el número no tiene decimales, se puede omitir la escala. Ejemplo: SALARIO NUMBER(7,2) define la columna SALARIO con 7 dígitos, 2 de ellos decimales. Los ejemplos mostrados en la Tabla 3.3 muestran el modo en que Oracle almacena los datos con diferentes precisiones y escalas.

Tabla 3.2. *Tipos de datos en Oracle 10g.*




CÓDIGO	Tipo de dato	Características
8	LONG	Almacena cadenas de caracteres de longitud variable que contengan hasta 2 gigabytes de información. Se puede usar este tipo para almacenar textos muy grandes. Ejemplo: TEXT LONG. Este tipo de dato está sujeto a algunas restricciones: sólo se puede definir una columna LONG por tabla, no pueden aparecer en restricciones de integridad (*constraints*), no sirve para indexar, una función almacenada no puede devolver un valor LONG, no se puede utilizar como argumento de funciones, no se puede referenciar como subconsulta en la creación de tablas ni inserción de filas, las variables o argumentos de bloques PL/SQL no se pueden declarar como tipo LONG, no es posible su uso en cláusulas WHERE, GROUP BY, ORDER BY, CONNECT BY o DISTINCT, ni con operaciones de UNION, INTERSECT y MINUS.
12	DATE	Almacena información de fechas y horas. Para cada tipo DATE se almacena la siguiente información: Siglo/Año/Mes/Día/Hora/Minutos/Segundos. Por omisión, el valor para el formato de la fecha se especifica con el parámetro **NLS_DATE_FORMAT**, y es una cadena de caracteres, como ésta 'DD/MM/YY', que representa día del mes/dos dígitos del mes/dos últimos dígitos del año. El formato de la fecha se puede cambiar mediante la orden ALTER SESSION y variando el parámetro NLS_DATE_FORMAT. Ejemplo: FECHA DATE.
23	RAW(tamaño)	Almacena datos binarios. Es similar al tipo VARCHAR2, con la diferencia de que maneja cadenas de bytes en lugar de cadenas de caracteres. El tamaño máximo es de 2.000 bytes.
24	LONG RAW	Almacena datos binarios. Es similar al tipo LONG; se emplea para el almacenamiento de gráficos, sonidos, etc. El tamaño máximo es de 2 gigabytes.
69	ROWID	Cadena hexadecimal que representa la dirección de una fila en su tabla.
1	NVARCHAR2(tamaño)	Tipo similar al VARCHAR2 solo que el tamaño de un carácter depende de la elección del juego de caracteres. El máximo tamaño es de 4.000 bytes.
96	NCHAR(tamaño)	Tipo similar al CHAR solo que el tamaño de un carácter depende de la elección del juego de caracteres. El máximo tamaño es de 2.000 bytes.
112	CLOB	Similar a LONG, se usa para objetos carácter. El tamaño máximo es de 4 gigabytes.
112	NCLOB	Similar al anterior solo que el tamaño del carácter depende del juego de caracteres.
113	BLOB	Similar a LONG RAW, se usa para objetos binarios. El tamaño máximo es de 4 gigabytes.

Tabla 3.2 ***(Continuación).*** *Tipos de datos en Oracle 10g.*

Dato actual	Formato	Almacenamiento
7456123.89	NUMBER	7456123.89
7456123.89	NUMBER(9)	7456124
7456123.89	NUMBER(9,2)	7456123.89
7456123.89	NUMBER(9,1)	7456123.9
7456123.8	NUMBER(6)	ERROR-Valor mayor que el que permite la precisión especificada para esta columna.
7456123.8	NUMBER(15,1)	7456123.8
7456123.89	NUMBER(7,-2)	7456100
7456123.89	NUMBER(-7,2)	ERROR-Especificador de precisión numérica está fuera de rango (1 a 38).

Tabla 3.3. *Datos numéricos, precisiones y escalas.*

3.5 SQL*Plus

Para aprender el lenguaje SQL nos serviremos del intérprete de sentencias SQL llamado **SQL*Plus**. Si deseamos empezar una sesión con SQL*Plus tenemos que conectarnos como usuarios Oracle; por tanto, el administrador de la base de datos deberá proporcionarnos un identificador de usuario con su *password* asociado. Este usuario dispondrá de una serie de derechos sobre los objetos de la base de datos. También deberá proporcionarnos la *Cadena de conexión* o *Cadena de Host* para conectarnos a la base de datos. Ésta se crea con la utilidad *Net Configuration Assistant* de Oracle. No hace falta escribir nada en este campo si la base de datos es local.

En una instalación típica podemos entrar haciendo *clic* en *Inicio/Programas/Oracle Developer Suite/Application Development/SQL*Plus*. O bien buscamos el archivo `sqlplusw.exe` y hacemos doble clic en él. Se abre una ventanita similar a la mostrada en la Figura 3.4, desde donde escribiremos el nombre de usuario, la contraseña y la cadena de conexión.

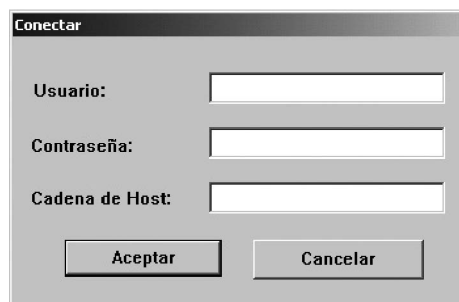


Figura 3.4. *Nos conectamos a Oracle.*

Una vez autenticados se visualiza una ventana similar a la mostrada en la Figura 3.5, desde la que veremos el *prompt* `SQL>` que nos indica que estamos conectados al gestor Oracle. Desde este momento podemos enviar sentencias SQL al gestor.

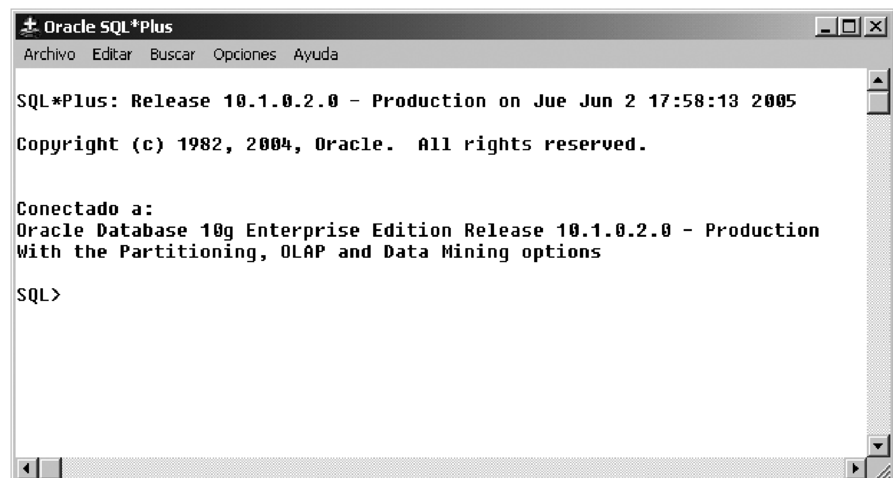


Figura 3.5. *Ventana Oracle SQL*Plus.*





Cuando trabajamos con SQL*Plus disponemos de un *buffer* de edición que contiene la última sentencia SQL que se intentó ejecutar. Mientras una sentencia se encuentre en el *buffer*, será posible modificarla a través de un conjunto de comandos de edición: `SQL> ED`. Este comando invocará al editor del sistema (por ejemplo, `NOTEPAD.EXE`), que abrirá el fichero asociado al *buffer* de edición (`afiedt.buf`), véase la Figura 3.6. Se puede usar para su modificación o archivado.

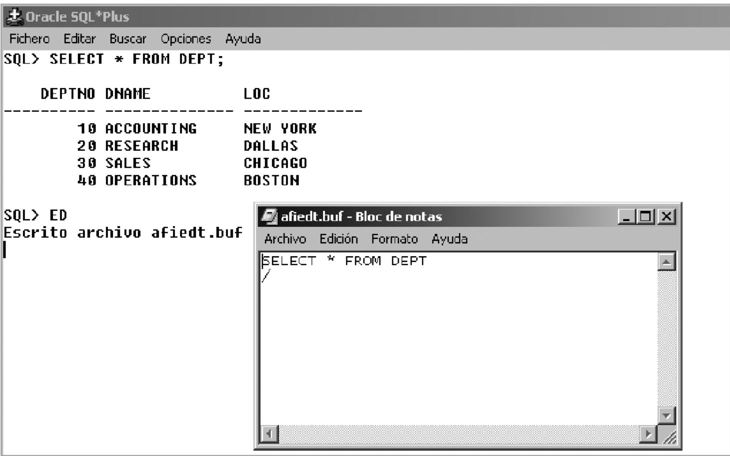


Figura 3.6. Buffer *de edición* `afiedt.buf`.

`SQL> LIST`	Visualiza el contenido del *buffer*. También podemos poner: `SQL> L`
`SQL> LIST n`	Se lista el número de línea *n*.
`SQL> SAVE fichero`	Almacena el contenido del *buffer* en *fichero.sql*.
`SQL> GET fichero`	Recupera en el *buffer* el contenido del archivo *fichero.sql*.
`SQL> START fichero`	Ejecuta el contenido almacenado en *fichero.sql*. Si el fichero no está en el directorio, tendríamos que poner todo el trayecto: `SQL> START c:\programas\fichero.sql`
`SQL> RUN`	Repite la ejecución de la última sentencia o de lo que hay en el *buffer*. También podemos poner: `SQL> R`
`SQL> INPUT`	Añade una línea a continuación de la actual activa.
`SQL> DEL`	Borra la línea actual.
`SQL> SPOOL fichero`	Todas las salidas por pantalla se almacenarán en un *fichero.lst*.
`SQL> SPOOL OFF`	Libera el almacenamiento de salidas por pantalla.
`SQL> CLEAR SCR`	Borra la pantalla.

Tabla 3.4. *Comandos que podemos usar desde el* prompt.

Actividades propuestas



1 **Conéctate como usuario Scott, contraseña Tiger y prueba las sentencias: `SELECT * FROM EMP;` y `SELECT * FROM DEPT;` y repasa todos los comandos vistos anteriormente. Anota lo que hace cada comando. Prueba las sentencias escribiendo cada palabra en una línea (es decir, escribimos `SELECT`, pulsamos la tecla *Enter*, escribimos *, pulsamos la tecla *Enter*, y así sucesivamente). ¿Qué observas?**

3.6 iSQL*Plus

En Oracle 9i se incluyó la versión web de SQL*Plus como componente adicional en la instalación de la base de datos. La versión 10g incluye este componente llamado **iSQL*Plus** que se instala en el servidor durante la instalación de la base de datos. Permite a los usuarios la ejecución de sentencias SQL y PL/SQL a través de un navegador web en el que se indica la dirección URL del servidor. Las salidas se generan en formato HTML con la posibilidad de guardarlas en archivos al igual que las sentencias que son mantenidas en un histórico durante la sesión.

Conexión y entorno de iSQL*Plus

Para conectarnos a iSQL*Plus abrimos un navegador web, por ejemplo Microsoft Internet Explorer y escribimos la siguiente URL: **http://nombredelservidor:5560/isqlplus**, donde *nombredeservidor* es el nombre de la máquina donde está instalada la base de datos. Una vez que se accede, el servicio solicita autenticación del usuario: nombre de usuario, clave y cadena de conexión.

Caso práctico

1 **Veamos cómo nos conectamos desde una máquina cliente a una máquina servidora de nombre SERVIDOR que tiene instalada una base de datos *Oracle 10g*.**

En primer lugar, abrimos el navegador y escribimos la siguiente URL: **http://servidor:5560/isqlplus.**

A continuación, escribimos el nombre de usuario: SCOTT, la contraseña TIGER y el identificador de conexión. Se muestra una imagen similar a la mostrada en la Figura 3.7. En este caso, el identificador de conexión coincide con el nombre de base de datos global o SID. Este dato lo debe suministrar el administrador de la base de datos.

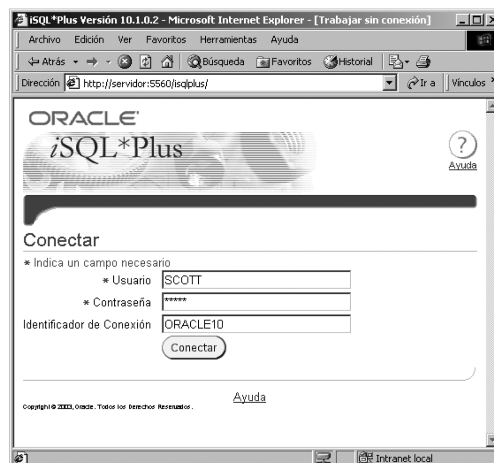


Figura 3.7. *Conexión iSQL*Plus.*

Se hace clic en el botón *Conectar*. Tras validarse correctamente la conexión aparece la pantalla de trabajo de iSQL*Plus.



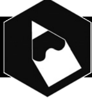
La pantalla de trabajo de iSQL*Plus presenta un área donde se escribirán las sentencias SQL, una serie de botones en la parte inferior y varios hipervínculos en la parte superior. Véase la Figura 3.8. Estos elementos se resumen en la Tabla 3.5.



Figura 3.8. *Espacio de trabajo iSQL*Plus.*

Elemento	Descripción
Ejecutar	Ejecuta la sentencia escrita en el espacio de trabajo, el resultado se muestra en la parte inferior debajo de los botones.
Cargar Archivo de Comandos	Muestra una nueva pantalla desde la que se puede cargar un archivo de comandos para ejecutarlo en el espacio de trabajo. Estos archivos tendrán la extensión **sql.**
Guardar Archivo de Comandos	Muestra un cuadro de diálogo donde hemos de dar un nombre al archivo que guardará lo escrito en el espacio de trabajo. Estos se guardan con extensión **sql.**
Cancelar	Cancela cualquier archivo de comandos que se esté ejecutando pero no limpia el espacio de trabajo.
Limpiar	Limpia el área de trabajo y todas las salidas mostradas.
Desconectar	Finaliza la sesión iSQL*Plus.
Preferencias	Abre una nueva pantalla desde la que se puede configurar los valores de la interfaz.
Ayuda	Abre la ayuda de iSQL*Plus en una ventana diferente.
Historial	Muestra una nueva pantalla desde la que se pueden ver y cargar las sentencias y archivos de comandos que se han ejecutado a lo largo de la sesión.

Tabla 3.5. *Elementos del espacio de trabajo iSQL*Plus.*



Caso práctico

2 **Desde el área de trabajo escribimos la siguiente orden: `SELECT * FROM DEPT`, y pulsamos el botón *Ejecutar*. En la parte inferior de la pantalla veremos el resultado de la ejecución, véase la Figura 3.9. Las órdenes SQL se pueden escribir en mayúscula o minúscula, colocando al final el punto y coma o sin punto y coma.**



Figura 3.9. *Ejecución de una sentencia desde iSQL*Plus.*

- Utiliza el botón *Guardar Archivo de Comandos* para guardar lo escrito en un archivo, por ejemplo en *myscript.sql*.
- Limpia el área de trabajo haciendo clic en el botón *Limpiar*.
- Haz clic en el botón *Cargar Archivo de Comandos*, y después clic en el botón *Examinar* para localizar el archivo guardado anteriormente y cargarle. Hacer clic en el botón *Cargar*. Se visualiza de nuevo el área de trabajo con el archivo de comandos.
- Haz clic en el botón *Ejecutar* para ejecutar la sentencia SQL. Se visualizan una serie de mensajes de error (ya que no son sentencias SQL ni PL/SQL) y, a continuación, el resultado de la sentencia SELECT.
- Limpia el área de trabajo y el área de salida haciendo clic en el botón *Limpiar* que está en la parte inferior derecha de la pantalla.
- Haz clic en *Historial*, se visualiza una nueva pantalla. Hacer clic en la sentencia SELECT, vemos que se carga en el área de trabajo. Ejecutar la sentencia de nuevo.
- Cierra la sesión haciendo clic en el botón *Desconectar*.



Actividades propuestas

2 **Prueba las sentencias: `SELECT * FROM EMP` y `SELECT * FROM DEPT` y repasa todos los conceptos vistos en el caso práctico anterior.**





3.7 Consulta de los datos

Para recuperar información o, lo que es lo mismo, para realizar consultas a la base de datos, utilizaremos una única sentencia SELECT. El usuario emplea esta sentencia con el nivel de complejidad apropiado para él: especifica qué es lo que quiere obtener, no dónde ni cómo.

De la consulta se puede obtener: cualquier unidad de datos, todos los datos, cualquier subconjunto de datos, cualquier conjunto de subconjuntos de datos.

> Todas las sentencias SQL terminan en punto y coma.

A. Sentencia SELECT

El formato de la sentencia SELECT es el siguiente:

```
SELECT [ALL|DISTINCT]
[expre_colum1, expre_colum2, ..., expre_column | * ]
FROM [nombre_tabla1,  nombre_tabla2, ..., nombre_tablan]
[WHERE condición]
[ORDER BY expre_colum [DESC|ASC] [,expre_colum
   [DESC|ASC]]...];
```

Donde `expre_colum` puede ser una columna de una tabla, una constante, una expresión aritmética, una función o varias funciones anidadas.

B. Cláusulas de SELECT

La única cláusula de la sentencia SELECT que es obligatoria es la cláusula FROM, el resto son opcionales.

> No importa si se usan mayúsculas o minúsculas.

- **FROM:** `FROM [nombre_tabla1, nombre_tabla2, ..., nombre_tablan]`

  Especifica la tabla o lista de tablas de las que se recuperarán los datos. Por ejemplo, consultamos los nombres de alumnos y su nota en la tabla ALUMNOS:

  ```
  SQL> SELECT NOM_ALUM, NOTA FROM ALUMNOS;
  ```

  Si el usuario que hace la consulta no es el propietario de la tabla, es necesario especificar el nombre de usuario delante de ella: *nombre_usuario.nombre_tabla*. Por ejemplo, si el usuario propietario de la tabla se llama PROFESOR emplearemos:

  ```
  SQL> SELECT NOM_ALUM, NOTA FROM PROFESOR.ALUMNOS;
  ```

  Es posible asociar un nuevo nombre a las tablas mediante *alias*. Por ejemplo, si la tabla ALUMNOS le damos el nombre de A, las columnas de la tabla iran acompañadas de A.

  ```
  SQL> SELECT A.NOM_ALUM, A.NOTA FROM ALUMNOS A;
  ```

- **WHERE:** `[WHERE condición]`

  Obtiene las filas que cumplen la condición expresada. La complejidad de la condición es prácticamente ilimitada. El formato de la condición es: *expresión operador expresión*.

  Las expresiones pueden ser una constante, una expresión aritmética, un valor nulo o un nombre de columna. Los operadores de comparación se pueden observar en el Tabla 3.6.

=, >, <, >=, <=, !=, <>
IN, NOT IN, BETWEEN, NOT BETWEEN
LIKE

Tabla 3.6. *Operadores de comparación.*

  Se pueden construir condiciones múltiples usando los operadores lógicos booleanos estándares: AND, OR y NOT. Está permitido emplear paréntesis para forzar el orden de evaluación. Veamos unos ejemplos de condiciones en la cláusula WHERE:

```
WHERE NOTA = 5
WHERE (NOTA>=10) AND (CURSO=1)
WHERE (NOTA IS NULL) OR (UPPER (NOM_ALUM) = 'PEDRO')
```

- **ORDER BY:** `[ORDER BY expre_columna [DESC|ASC] [,expre columna [DESC|ASC]]...]`

  Esta cláusula especifica el criterio de clasificación del resultado de la consulta. ASC especifica una ordenación ascendente, y DESC descendente.

  Puede contener expresiones con valores de columnas, por ejemplo:

```
SQL> SELECT * FROM ALUMNOS ORDER BY NOTA/5;
```

  Asimismo, es posible anidar los criterios. El situado más a la izquierda será el principal. Por ejemplo: `SQL> SELECT * FROM ALUMNOS ORDER BY NOM_ALUM, CURSO DESC;` (ordena por NOM_ALUM ascendente y por CURSO descendente).

  También se puede indicar mediante un número, que indica la posición de la columna a la derecha de SELECT, el criterio de clasificación. Por ejemplo: `SQL>SELECT DEPT_NO, DNOMBRE, LOC FROM DEPART ORDER BY 2;` ordena la salida por la segunda columna que es DNOMBRE.

DEPT_NO
——————
20
30
30
20
30
30
10
20
10
30
20
30
20
10
14 filas seleccionadas.

Tabla 3.7. *Salida sin DISTINCT.*

- **ALL:** Con la claúsula ALL recuperamos todas las filas, aunque algunas estén repetidas. Es la opción por omisión.

- **DISTINCT:** Sólo recupera las filas que son distintas. Por ejemplo, consultamos los departamentos (columna DEPT_NO) de la tabla EMPLE. El resultado lo podemos observar en la Tabla 3.7: `SQL> SELECT DEPT_NO FROM EMPLE;`

  Aparecen todas las filas de la tabla EMPLE donde la columna DEPT_NO no sea nula; también aparecen números de departamento repetidos. Con DISTINCT se eliminan las filas repetidas, como podemos observar en la Tabla 3.8: `SQL> SELECT DISTINCT DEPT_NO FROM EMPLE;`

DEPT_NO
——————
10
20
30

Tabla 3.8. *Salida con DISTINCT.*





C. Selección de columnas

El formato de SELECT que nos permite seleccionar las columnas de una tabla es éste:

```
SELECT [ALL|DISTINCT]
      [expre_colum1, expre_colum2, ..., expre_column | * ]
FROM  [nombre_tabla1, nombre_tabla2, ..., nombre_tablan]
```

A modo de ejemplo, vamos a realizar consultas sobre las tablas EMPLE y DEPART.

Veamos antes la descripción de estas tablas con la orden **DESC**ribe, que nos da un resumen de la tabla y de sus columnas:

```
SQL> DESC EMPLE
  Nombre                         ¿Nulo?     Tipo
  -----------------------------  --------   ---------
  EMP_NO                         NOT NULL   NUMBER(4)
  APELLIDO                                  VARCHAR2(10)
  OFICIO                                    VARCHAR2(10)
  DIR                                       NUMBER(4)
  FECHA_ALT                                 DATE
  SALARIO                                   NUMBER(7)
  COMISION                                  NUMBER(7)
  DEPT_NO                        NOT NULL   NUMBER(2)

SQL> DESC DEPART
  Nombre                         ¿Nulo?     Tipo
  -----------------------------  --------   ---------
  DEPT_NO                        NOT NULL   NUMBER(2)
  DNOMBRE                                   VARCHAR2(14)
  LOC                                       VARCHAR2(14)
```

Los *scripts* de creación de las tablas para esta unidad se encuentran en el CD- ROM.

NOT NULL	Significa que la columna no puede tener valores nulos.
VARCHAR2	Tipo de datos cadena de longitud variable. Puede contener cualquier carácter. La longitud máxima de la cadena se define entre paréntesis.
DATE	Tipo de datos fecha.
NUMBER	Tipo de datos numérico. El número de dígitos se expresa entre paréntesis.

Tabla 3.9. *Descripción de los tipos.*

En cuanto a la selección de todas las columnas de la tabla EMPLE, podemos recuperar las filas de dos formas:

1. Ponemos los nombres de todas las columnas, uno tras otro, separados por comas: `SQL> SELECT EMP_NO, APELLIDO, OFICIO, DIR, FECHA_ALT, SALARIO, COMISION, DEPT_NO  FROM EMPLE;`

2. Ponemos *, que representa a todas las columnas de la tabla: `SQL> SELECT * FROM EMPLE;`

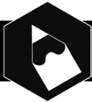
Para seleccionar determinadas columnas, sólo se ponen los nombres de las columnas que nos interesan. Por ejemplo, para escoger el nombre (DNOMBRE) y el número de departamento (DEPT_NO) de la tabla DEPART emplearemos:

```
SQL> SELECT DNOMBRE, DEPT_NO FROM DEPART;

            DNOMBRE           DEPT_NO
            -------------- ----------
            CONTABILIDAD           10
            INVESTIGACION          20
            VENTAS                 30
            PRODUCCION             40
```

D. Selección por fila

Para seleccionar determinadas filas necesitamos emplear la cláusula WHERE en la sentencia SELECT, `[WHERE condición]`.



Caso práctico

3 **Seleccionamos de la tabla EMPLE a todos los empleados del departamento 20 (DEPT_NO =20). Además, la consulta debe aparecer ordenada por la columna APELLIDO. Los campos que hay que consultar son: número de empleado, apellido, oficio y número de departamento:**

```
SQL> SELECT EMP_NO, APELLIDO, OFICIO, DEPT_NO FROM EMPLE WHERE DEPT_NO = 20 ORDER
BY APELLIDO;

            EMP_NO    APELLIDO      OFICIO          DEPT_NO
         ---------    ----------    ----------   ----------
              7876    ALONSO        EMPLEADO             20
              7902    FERNANDEZ     ANALISTA             20
              7788    GIL           ANALISTA             20
              7566    JIMENEZ       DIRECTOR             20
              7369    SANCHEZ       EMPLEADO             20
```

Consulta de los empleados cuyo oficio sea 'ANALISTA' ordenado por número de empleado (columna EMP_NO):

```
SQL> SELECT  *  FROM EMPLE WHERE  OFICIO = 'ANALISTA' ORDER  BY  EMP_NO;
```

Seleccionar de la tabla EMPLE aquellas filas del departamento 10 y cuyo oficio sea 'ANALISTA'. La consulta se ha de ordenar de modo descendente por APELLIDO y también de manera descendente por número de empleado (columna EMP_NO):

```
SQL> SELECT  *  FROM EMPLE WHERE  DEPT_NO=10 AND OFICIO = 'ANALISTA' ORDER  BY  APELLIDO
DESC, EMP_NO DESC;
```





Actividades propuestas



3 A partir de la tabla ALUM0405 (véase Tabla 3.10) que contiene los datos de alumnos matriculados en el curso 2004/2005 para un centro de enseñanza:

Columna	Tipo de dato	Descripción
DNI	VARCHAR2(10)	DNI del alumno
NOMBRE	VARCHAR2(15)	Nombre del alumno
APELLIDOS	VARCHAR2(20)	Apellidos del alumno
FECHA_NAC	DATE	Fecha de nacimiento
DIRECCION	VARCHAR2(20)	Dirección del alumno
POBLACION	VARCHAR2(20)	Población del alumno
PROVINCIA	VARCHAR2(20)	Provincia del alumno
CURSO	NUMBER(1)	Curso del alumno (1, 2, 3, 4)
NIVEL	VARCHAR2(3)	Nivel (ESO, BAC, DAI, ASI, ADM, COM)
CLASE	CHAR(2)	Aula en la que está el alumno
FALTAS1	NUMBER(2)	Faltas primer trimestre
FALTAS2	NUMBER(2)	Faltas segundo trimestre
FALTAS3	NUMBER(2)	Faltas tercer trimestre

Tabla 3.10. *Tabla ALUM0405.*

- Obtén todos los datos de los alumnos.
- Obtén los siguientes datos de alumnos: DNI, NOMBRE, APELLIDOS, CURSO, NIVEL y CLASE.
- Obtén todos los datos de alumnos cuya población sea 'GUADALAJARA'.
- Obtén el NOMBRE y APELLIDOS de todos los alumnos cuya población sea 'GUADALAJARA'.
- Consulta el DNI, NOMBRE, APELLIDOS, CURSO, NIVEL y CLASE de todos los alumnos ordenado por APELLIDOS y NOMBRE ascendentemente.

E. Crear y utilizar alias de columnas

Cuando se consulta la base de datos, los nombres de las columnas se usan como cabeceras de presentación. Si el nombre resulta demasiado largo, corto o críptico, existe la posibilidad de cambiarlo con la misma sentencia SQL de consulta creando un ALIAS. El ALIAS se pone entre comillas dobles, a la derecha de la columna.


Caso práctico

4 **Utilizamos alias para las columnas DEPT_NO y DNOMBRE:**

```
SQL> SELECT DNOMBRE "Departamento", DEPT_NO "Número Departamento" FROM DEPART;

                      Departamento    Número Departamento
                      --------------  -------------------
                      CONTABILIDAD                     10
                      INVESTIGACION                    20
                      VENTAS                           30
                      PRODUCCION                       40
```

3.8 Operadores aritméticos

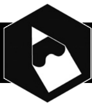
Operador aritmético	Operación
+	Suma
-	Resta
*	Multiplicación
/	División

Tabla 3.11. *Operadores aritméticos.*

Los **operadores aritméticos** sirven para formar expresiones con constantes, valores de columnas y funciones de valores de columnas. Son los que podemos observar en el Tabla 3.11.

Una forma posible de utilizar los operadores puede ser:

```
SELECT col1*col2, col1-col2  FROM  tabla1 WHERE col1+col2 = 34;
```


Caso práctico

5 **Disponemos de la tabla NOTAS_ALUMNOS, que contiene las notas de los alumnos de primer curso de ciclo DAI obtenidas en los tres módulos. La descripción de la tabla es la siguiente:**

```
SQL> DESC NOTAS_ALUMNOS
Nombre                                          ¿Nulo?      Tipo
----------------------------------------------  --------    -------------
NOMBRE_ALUMNO                                   NOT NULL    VARCHAR2(25)
NOTA1                                                       NUMBER(2)
NOTA2                                                       NUMBER(2)
NOTA3                                                       NUMBER(2)
```

(Continúa)





(Continuación)

```
SQL> SELECT * FROM NOTAS_ALUMNOS;

NOMBRE_ALUMNO                    NOTA1      NOTA2      NOTA3
-------------------------    ----------  ---------- ----------
Alcalde García, M. Luisa             5          5          5
Benito Martín, Luis                  7          6          8
Casas Martínez, Manuel               7          5          5
Corregidor Sánchez, Ana              6          9          8
```

Se trata de obtener la nota media de cada alumno. Visualizamos por cada uno de ellos su nombre y su nota media (suma de las tres notas dividida entre tres):

```
SQL> SELECT NOMBRE_ALUMNO "Nombre Alumno", (NOTA1+NOTA2+NOTA3)/3 "Nota Media" FROM
NOTAS_ALUMNOS;

Nombre Alumno                Nota Media
-------------------------    ----------
Alcalde García, M. Luisa              5
Benito Martín, Luis                   7
Casas Martínez, Manuel       5,66666667
Corregidor Sánchez, Ana      7,66666667
```

3.9 Operadores de comparación y lógicos

Los operadores de comparación y los operadores lógicos, se resumen en las Tablas 3.12 y 3.13.

Operador	Función
=	Igual a
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
!= <>	Distinto de

Tabla 3.12. *Operadores de comparación.*

Operador	Función
AND	Devuelve el valor TRUE cuando las dos condiciones son verdaderas.
OR	Devuelve el valor TRUE cuando una de las dos condiciones es verdadera.
NOT	Devuelve el valor TRUE si la condición es falsa.

Tabla 3.13. *Operadores lógicos.*



Caso práctico

6 **A partir de la tabla NOTAS_ALUMNOS, deseamos obtener aquellos nombres de alumnos que tengan un 7 en NOTA1 y cuya media sea mayor que 6:**

```
SQL> SELECT NOMBRE_ALUMNO FROM NOTAS_ALUMNOS WHERE NOTA1=7 AND (NOTA1+NOTA2+NOTA3)/3 >6;

NOMBRE_ALUMNO
-------------------------
Benito Martín, Luis
```

3.10 Operadores de comparación de cadenas de caracteres

Para comparar cadenas de caracteres, hasta ahora hemos utilizado el operador de comparación Igual a (=). Así, a partir de la tabla EMPLE, obtenemos el apellido de los ANALISTAS del departamento 10:

```
SQL> SELECT  APELLIDO  FROM EMPLE WHERE  OFICIO = 'ANA-
LISTA' AND DEPT_NO=10;
```

Pero este operador no nos sirve si queremos hacer consultas de este tipo:

Obtener los datos de los empleados cuyo apellido empiece por una «P» u «obtener los nombres de alumnos que incluyan la palabra *Pérez*».

Para especificar este tipo de consultas, en SQL usamos el operador LIKE que permite utilizar los siguientes caracteres especiales en las cadenas de comparación:

% Comodín: representa cualquier cadena de 0 o más caracteres.

'_' Marcador de posición: representa un carácter cualquiera.

En la cláusula WHERE este operador se utilizará de la siguiente manera:

```
WHERE columna LIKE 'caracteres_especiales'
```

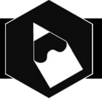
En una sentencia WHERE se pueden usar varias cláusulas LIKE anidadas por operadores AND/OR:

```
WHERE col1 LIKE 'car_especiales' AND|OR col2 LIKE
'car_especiales'.
```





Caso práctico



7 A continuación se muestran ejemplos de uso de la cláusula LIKE:

`LIKE 'Director'`	la cadena 'Director'.
`LIKE 'M%'`	cualquier cadena que empiece por 'M'.
`LIKE '%X%'`	cualquier cadena que contenga una 'X'.
`LIKE '__M'`	cualquier cadena de 3 caracteres terminada en 'M'.
`LIKE 'N_'`	una cadena de 2 caracteres que empiece por 'N'.
`LIKE '_R%'`	cualquier cadena cuyo segundo carácter sea una 'R'.

Hemos de tener en cuenta que las mayúsculas y minúsculas son significativas ('m' no es lo mismo que 'M') y que las constantes alfanuméricas deben encerrarse siempre entre comillas simples.

– A partir de la tabla EMPLE, obtén aquellos apellidos que empiecen por una 'J':

```
SQL> SELECT APELLIDO FROM EMPLE WHERE APELLIDO LIKE 'J%';
APELLIDO
----------
JIMENEZ
JIMENO
```

– Obtén aquellos apellidos que tengan una 'R' en la segunda posición:

```
SQL> SELECT APELLIDO FROM EMPLE WHERE APELLIDO LIKE '_R%';
APELLIDO
----------
ARROYO
```

– Obtén aquellos apellidos que empiecen por 'A' y tengan una 'O' en su interior:

```
SQL> SELECT APELLIDO FROM EMPLE WHERE APELLIDO LIKE 'A%O%';
APELLIDO
----------
ARROYO
ALONSO
```

A continuación, consideramos la tabla LIBRERÍA, cuya descripción es la siguiente:

```
SQL> DESC LIBRERIA
Nombre                                              ¿Nulo?       Tipo
-------------------------------------------------   --------     ----------
TEMA                                                NOT NULL     CHAR(15)
ESTANTE                                                          CHAR(1)
EJEMPLARES                                                       NUMBER(2)
```

(Continúa)

(Continuación)

– Consultamos las filas de la tabla LIBRERIA cuyo tema sea 'LABORES'; usamos el operador ' = ':

```
SQL> SELECT * FROM LIBRERIA WHERE TEMA='LABORES';
TEMA           E EJEMPLARES
--------------- - ----------
```

– Hacemos lo mismo, pero ahora manejando el operador LIKE:

```
SQL> SELECT * FROM LIBRERIA WHERE TEMA LIKE 'LABORES';

ninguna fila seleccionada
```

La consulta no devuelve nada, debido a que la columna TEMA está definida con el tipo CHAR(15). El tipo CHAR rellena blancos a la derecha hasta formar la longitud de 15 caracteres. Para que funcione la consulta tendremos que utilizar el comodín %:

```
SQL>  SELECT * FROM LIBRERIA WHERE TEMA LIKE 'LABORES%';

 TEMA              E   EJEMPLARES
 ---------------   -   ----------
 LABORES           B           20
```

Si la columna TEMA fuese de tipo VARCHAR2, no sería necesario usar el comodín % con el operador LIKE.

3.11 NULL y NOT NULL

Se dice que una columna de una fila es NULL si está completamente vacía. Para comprobar si el valor de una columna es nulo empleamos la expresión: *columna* ***IS NULL***. Si queremos saber si el valor de una columna no es nulo, usamos la expresión: *columna* ***IS NOT NULL***. Cuando comparamos con valores nulos o no nulos no podemos utilizar los operadores de igualdad, mayor o menor.

Por ejemplo, a partir de la tabla EMPLE, consultamos los apellidos de los empleados cuya comisión es nula:

```
SQL> SELECT APELLIDO FROM EMPLE WHERE COMISION IS NULL;
```

Si queremos consultar los apellidos de los empleados cuya comisión no sea nula teclearemos esto:

```
SQL> SELECT APELLIDO FROM EMPLE WHERE COMISION IS NOT NULL;
```





3.12 Comprobaciones con conjuntos de valores

Hasta ahora, todas las comprobaciones lógicas que hemos visto comparan una columna o expresión con un valor. Por ejemplo:

```
OFICIO = 'ANALISTA' AND DEPT_NO=10.
```

Pero también podemos comparar una columna o una expresión con una lista de valores utilizando los operadores **IN** y **BETWEEN**.

A. Operador IN

El operador **IN** nos permite comprobar si una expresión pertenece o no (NOT) a un conjunto de valores, haciendo posible la realización de comparaciones múltiples. Su formato es:

```
<expresión> [NOT] IN (lista de valores separados por comas)
```

Caso práctico



8 La lista de valores está formada por números:

– Consulta los apellidos de la tabla EMPLE cuyo número de departamento sea 10 o 30:

```
SQL> SELECT APELLIDO FROM EMPLE WHERE DEPT_NO IN(10,30);
```

– Consulta los apellidos de la tabla EMPLE cuyo número de departamento no sea ni 10 ni 30:

```
SQL> SELECT APELLIDO FROM EMPLE WHERE DEPT_NO NOT IN(10,30);
```

La lista de valores está formada por cadenas:

– Consulta los apellidos de la tabla EMPLE cuyo oficio sea 'VENDEDOR', 'ANALISTA' o 'EMPLEADO':

```
SQL> SELECT APELLIDO FROM EMPLE WHERE OFICIO IN ('VENDEDOR', 'ANALISTA',
'EMPLEADO');
```

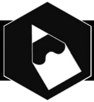
– Consulta los apellidos de la tabla EMPLE cuyo oficio no sea ni 'VENDEDOR' ni 'ANALISTA' ni 'EMPLEADO':

```
SQL> SELECT APELLIDO FROM EMPLE WHERE OFICIO NOT IN ('VENDEDOR', 'ANALISTA',
'EMPLEADO');
```

B. Operador BETWEEN

El operador **BETWEEN** comprueba si un valor está comprendido o no (NOT) dentro de un rango de valores, desde un valor inicial a un valor final. Su formato es:

```
<expresión> [NOT] BETWEEN valor_inicial AND valor_final
```



Caso práctico

9 **A partir de la tabla EMPLE, obtén el apellido y el salario de los empleados cuyo salario esté comprendido entre 1500 y 2000:**

```
SQL> SELECT APELLIDO, SALARIO FROM EMPLE WHERE SALARIO BETWEEN 1500 AND 2000;

                              APELLIDO        SALARIO
                              ----------   ----------
                              ARROYO             1500
                              SALA               1625
                              MARTIN             1600
                              MUÑOZ              1690
```

En esta sentencia se visualizan los empleados cuyo salario es 1500, 2000 o cualquier valor comprendido entre ambos.

– A partir de la tabla EMPLE, obtener el apellido y el salario de los empleados cuyo SALARIO no esté comprendido entre 1500 y 2000:

```
SQL> SELECT APELLIDO, SALARIO FROM EMPLE WHERE SALARIO NOT BETWEEN 1500 AND 2000;

     APELLIDO       SALARIO
     ---------- ----------
     SANCHEZ           1040             REY            4100
     JIMENEZ           2900             TOVAR          1350
     NEGRO             3005             ALONSO         1430
     CEREZO            2885             JIMENO         1335
     GIL               3000             FERNANDEZ      2900

10 filas seleccionadas.
```

Con esta sentencia tenemos los empleados cuyo SALARIO es menor que 1500 o mayor que 2000.

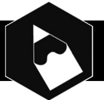



3.13 Combinación de operadores AND y OR

Los operadores AND y OR se pueden combinar de forma ilimitada, pero hay que tener cuidado al usarlos y utilizar paréntesis para agrupar aquellas expresiones que se deseen evaluar juntas; si no nos servimos de los paréntesis, es posible que los resultados no sean los deseados.

El orden de prioridad de los operadores lógicos es el siguiente: el primero NOT, después AND y, por último, OR. Para alterar el orden se utilizan paréntesis.

Caso práctico



10 **A partir de la tabla EMPLE, obtén el APELLIDO, SALARIO y DEPT_NO de los empleados cuyo salario sea mayor de 2000 en los departamentos 10 o 20:**

```
SQL> SELECT APELLIDO, SALARIO, DEPT_NO  FROM EMPLE WHERE
SALARIO >2000 AND (DEPT_NO=10 OR DEPT_NO=20);

APELLIDO       SALARIO     DEPT_NO
----------  ----------  ----------
JIMENEZ           2900          20
CEREZO            2885          10
GIL               3000          20
REY               4100          10
FERNANDEZ         2900          20
```

Sin utilizar paréntesis, la consulta anterior dará el siguiente resultado:

```
SQL> SELECT APELLIDO, SALARIO, DEPT_NO FROM EMPLE WHERE SALARIO >2000 AND
DEPT_NO=10 OR DEPT_NO=20;

APELLIDO       SALARIO     DEPT_NO
----------  ----------  ----------
SANCHEZ           1040          20
JIMENEZ           2900          20
CEREZO            2885          10
GIL               3000          20
REY               4100          10
ALONSO            1430          20
FERNANDEZ         2900          20

7 filas seleccionadas.
```

En el ejemplo, al no usar paréntesis se obtienen los empleados cuyo SALARIO sea > 2000 y el DEPT_NO = 10; o bien, aquellos cuyo DEPT_NO sea 20.

– La consulta inicial también se puede hacer recurriendo al operador IN:

```
SQL> SELECT APELLIDO, SALARIO, DEPT_NO FROM EMPLE WHERE SALARIO >2000 AND DEPT_NO
IN(10,20);
```

3.14 Subconsultas

A veces, para realizar alguna operación de consulta, necesitamos los datos devueltos por otra consulta; así, si queremos obtener los datos de los empleados que tengan el mismo oficio que 'PEPE', hemos de averiguar el oficio de 'PEPE' (primera consulta). Una vez conocido este dato, podemos averiguar qué empleados tienen el mismo oficio que 'PEPE' (segunda consulta). Este problema se puede resolver usando una subconsulta, que no es más que una sentencia SELECT dentro de otra SELECT. Las **subconsultas** son aquellas sentencias SELECT que forman parte de una cláusula WHERE de una sentencia SELECT anterior. Una subconsulta consistirá en incluir una declaración SELECT como parte de una cláusula WHERE. El formato de una subconsulta es similar a éste:

```
SELECT  ...
FROM    ...
WHERE  columna  operador_comparativo   (SELECT ...
                                        FROM ...
                                        WHERE ... );
```

La subconsulta (el comando SELECT entre paréntesis) se ejecutará primero y, posteriormente, el valor extraído es «introducido» en la consulta principal.



Caso práctico

11 **Con la tabla EMPLE, obtén el APELLIDO de los empleados con el mismo OFICIO que 'GIL'. Para ello, descomponemos el enunciado en dos consultas. Primero averiguamos el OFICIO de 'GIL':**

```
SQL> SELECT OFICIO FROM EMPLE WHERE APELLIDO ='GIL';

OFICIO
----------
ANALISTA
```

– A continuación, visualizamos el APELLIDO de los empleados con el mismo OFICIO que 'GIL':

```
SQL> SELECT APELLIDO FROM EMPLE WHERE OFICIO='ANALISTA';

APELLIDO
----------
GIL
FERNANDEZ
```

– Es posible resumir estas dos consultas con una sentencia SELECT que forma parte de la cláusula WHERE:

```
SQL> SELECT APELLIDO FROM EMPLE WHERE OFICIO = (SELECT OFICIO FROM EMPLE WHERE APE-
LLIDO ='GIL');

APELLIDO
----------
GIL
FERNANDEZ
```




Actividades propuestas



4 **Muestra los datos (apellido, oficio, salario y fecha de alta) de aquellos empleados que desempeñen el mismo oficio que "JIMENEZ" o que tengan un salario mayor o igual que "FERNANDEZ".**

A. Condiciones de búsquedas en subconsultas

Las **subconsultas** normalmente aparecen como parte de la condición de búsqueda de una cláusula WHERE o HAVING. Las condiciones de búsqueda que nos podemos encontrar en una subconsulta son:

- **Test de comparación en subconsultas (>, <, <>, <=, >=, =).** Compara el valor de una expresión con un valor único producido por una subconsulta. Ejemplo: Obtener aquellos apellidos de empleados cuyo oficio es igual al oficio de 'GIL':

```
SELECT APELLIDO FROM EMPLE WHERE OFICIO = (SELECT OFICIO
FROM EMPLE WHERE APELLIDO ='GIL');
```

- **Test de pertenencia a un conjunto devuelto por una subconsulta (IN).** Comprueba si el valor de una expresión coincide con uno del conjunto de valores producido por una subconsulta. Ejemplo: Obtener aquellos apellidos de empleados cuyo oficio sea alguno de los oficios que hay en el departamento 20:

```
SELECT APELLIDO FROM EMPLE WHERE OFICIO IN (SELECT OFICIO
FROM EMPLE WHERE DEPT_NO=20);
```

- **Test de existencia (EXISTS , NOT EXISTS).** Examina si una subconsulta produce alguna fila de resultados. El test es TRUE si devuelve filas, si no es FALSE. Ejemplo: Listar los departamentos que tengan empleados:

```
SELECT DNOMBRE, DEPT_NO FROM DEPART WHERE EXISTS (SELECT
* FROM EMPLE WHERE EMPLE.DEPT_NO= DEPART.DEPT_NO);
```

  Para calcular los que no tengan empleados se usa el operador NOT EXISTS.

- **Test de comparación cuantificada (ANY y ALL).** Se usan en conjunción con los operadores de comparación (>, <, <>, <=, >=, =).

  ANY compara el valor de una expresión con cada uno del conjunto de valores producido por una subconsulta, si alguna de las comparaciones individuales da como resultado TRUE, ANY devuelve TRUE, si la subconsulta no devuelve nada devolverá FALSE. Ejemplo: obtener los datos de los empleados cuyo salario sea igual a algún salario de los empleados del departamento 30:

```
SELECT * FROM EMPLE WHERE SALARIO = ANY  (SELECT SALARIO
FROM EMPLE WHERE DEPT_NO=30);
```

ALL compara el valor de una expresión con cada uno del conjunto de valores producido por una subconsulta, si todas las comparaciones individuales da como resultado TRUE, ALL devuelve TRUE, en caso contrario devuelve FALSE. Ejemplo: Obtener los datos de los empleados cuyo salario sea menor a cualquier salario de los empleados del departamento 30:

```
SELECT * FROM EMPLE WHERE SALARIO < ALL (SELECT SALARIO
FROM EMPLE WHERE DEPT_NO=30);
```

B. Subconsultas que generan valores simples

Se trata de aquellas subconsultas que devuelven una fila o un valor simple; en éstas se utilizan los operadores de comparación (>, <, <>, <=, >=, =). Si la subconsulta obtiene más de una fila, se produce un mensaje de error. Por ejemplo, con la siguiente consulta se pretende obtener los apellidos de los empleados cuyo oficio coincida con algún oficio del departamento 20:

```
SQL>  SELECT APELLIDO FROM EMPLE WHERE OFICIO =
  2   (SELECT OFICIO FROM EMPLE where DEPT_NO=20);
 (SELECT OFICIO FROM EMPLE where DEPT_NO=20)
  *
ERROR en línea 2:
ORA-01427: la subconsulta de una sola fila devuelve más
de una fila
```

En el departamento 20 hay varios oficios por tanto la subconsulta devuelve varias filas. Por ello, cuando una subconsulta devuelve más de una fila no se puede recurrir a los operadores de comparación.

C. Subconsultas que generan listas de valores

Son aquellas subconsultas que devuelven más de una fila o más de un valor. Cuando una subconsulta devuelva más de un valor, usaremos el operador IN en la cláusula WHERE. La solución al ejemplo anterior sería la siguiente:

```
SQL> SELECT APELLIDO FROM EMPLE WHERE OFICIO IN
(SELECT OFICIO FROM EMPLE where DEPT_NO=20);
```

El tipo de dato de la expresión situada después de WHERE debe coincidir con el tipo de dato devuelto por la subconsulta.




Caso práctico



12 **Usamos las tablas EMPLE y DEPART. Queremos consultar los datos de los empleados que trabajen en 'MADRID' o 'BARCELONA'. La localidad de los departamentos se obtiene de la tabla DEPART. Hemos de relacionar las tablas EMPLE y DEPART por el número de departamento. Para ello, descomponemos esa consulta en varias: primero tenemos que localizar los números de departamento que estén en la localidad de 'MADRID' o 'BARCELONA':**

```
SELECT DEPT_NO FROM DEPART WHERE LOC IN ('MADRID', 'BARCELONA');

         DEPT_NO
      ----------
              20
              30
```

A continuación, seleccionamos los datos de los empleados que estén en esos departamentos:

```
SELECT EMP_NO, APELLIDO, OFICIO, DIR, FECHA_ALT, SALARIO, COMISION, DEPT_NO FROM
EMPLE  WHERE  DEPT_NO  IN(20, 30);
```

Por último, reunimos estas dos sentencias SELECT utilizando una subconsulta:

```
SELECT EMP_NO, APELLIDO, OFICIO, DIR, FECHA_ALT, SALARIO, COMISION, DEPT_NO
FROM  EMPLE  WHERE  DEPT_NO  IN  (SELECT  DEPT_NO  FROM  DEPART  WHERE  LOC  IN
('MADRID','BARCELONA'));
```

– Consulta los apellidos y oficios de todos los empleados del departamento 20 cuyo trabajo sea idéntico al de cualquiera de los empleados del departamento 'VENTAS':

```
SQL> SELECT APELLIDO, OFICIO FROM EMPLE WHERE DEPT_NO=20 AND
  2  OFICIO IN (SELECT OFICIO FROM EMPLE WHERE DEPT_NO =
  3  (SELECT  DEPT_NO  FROM  DEPART  WHERE DNOMBRE='VENTAS'));

         APELLIDO     OFICIO
         ----------   ----------
         SANCHEZ      EMPLEADO
         JIMENEZ      DIRECTOR
         ALONSO       EMPLEADO
```

– Obtén el apellido de los empleados con el mismo oficio y salario que 'GIL'. En esta consulta se introduce una variante: hasta ahora las subconsultas nos devolvían una columna, aunque pueden devolver más de una. En este caso, la subconsulta devuelve dos columnas, el oficio y el salario:

```
SQL> SELECT APELLIDO, SALARIO FROM EMPLE WHERE (OFICIO, SALARIO) =
  2  (SELECT OFICIO, SALARIO FROM EMPLE WHERE APELLIDO ='GIL');

         APELLIDO        SALARIO
         ----------   ----------
         GIL                3000
         FERNANDEZ          3000
```

(Continúa)

(Continuación)

Si se detallan varios campos en la cláusula WHERE, deben encerrarse entre paréntesis, y han de coincidir en número y tipo de datos con los de la sentencia SELECT de la consulta interior. Cada columna de la cláusula WHERE se referirá a la correspondiente columna de la subconsulta. También podíamos haber puesto esto:

```
SQL> SELECT APELLIDO, SALARIO FROM EMPLE
  2  WHERE OFICIO= (SELECT OFICIO FROM EMPLE WHERE APELLIDO ='GIL')
  3  AND SALARIO= (SELECT SALARIO  FROM EMPLE WHERE APELLIDO ='GIL');
```



Actividades propuestas

5 **Presenta los apellidos y oficios de los empleados que tienen el mismo trabajo que "JIMENEZ".**

Muestra en pantalla el APELLIDO, OFICIO y SALARIO de los empleados del departamento de "FERNANDEZ" que tengan su mismo salario.

D. Subconsultas correlacionadas

Una **subconsulta correlacionada** es aquella que hace referencia a una columna o varias de la consulta más externa. A veces la subconsulta hace uso de columnas que tienen el mismo nombre que las columnas de las tablas usadas en la consulta mas externa. Si la subconsulta necesita acceder a esas columnas deberá definirse un alias en la tabla más externa. Por ejemplo, deseamos obtener los datos de los empleados cuyo salario sea el máximo salario de su departamento:

```
SELECT * FROM EMPLE E WHERE SALARIO = (SELECT
MAX(SALARIO) FROM EMPLE WHERE DEPT_NO = E.DEPT_NO);
```

La subconsulta devuelve para cada fila que se recupere de la consulta más externa el máximo salario del departamento que se está recuperando en la consulta externa; para referenciar a dicho departamento se necesita el alias **E** usado en la tabla de la consulta externa.

3.15 Combinación de tablas

Hasta ahora, en las consultas que hemos realizado sólo se ha utilizado una tabla, indicada a la derecha de la palabra FROM; pero hay veces que una consulta necesita columnas de varias tablas. En este caso, las tablas se expresarán a la derecha de la palabra FROM.



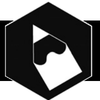

Sintaxis general:

```
SELECT   columnas de las tablas citadas en la cláusula
         "from"
FROM     tabla1, tabla2,...
WHERE    tabla1.columna = tabla2.columna;
```

Cuando combinamos varias tablas, hemos de tener en cuenta una serie de reglas:

- Es posible unir tantas tablas como deseemos.
- En la cláusula SELECT se pueden citar columnas de todas las tablas.
- Si hay columnas con el mismo nombre en las distintas tablas de la cláusula FROM, se deben identificar, especificando `NombreTabla.NombreColumna`.
- Si el nombre de una columna existe sólo en una tabla, no será necesario especificarla como `NombreTabla.NombreColumna`. Sin embargo, hacerlo mejoraría la legibilidad de la sentencia SELECT.
- El criterio que se siga para combinar las tablas ha de especificarse en la cláusula WHERE. Si se omite esta cláusula, que especifica la condición de combinación, el resultado será un PRODUCTO CARTESIANO, que emparejará todas las filas de una tabla con cada fila de la otra.

Caso práctico



13 **A partir de las tablas EMPLE y DEPART obtenemos los siguientes datos de los empleados: APELLIDO, OFICIO, número de empleado (EMP_NO), nombre de departamento (DNOMBRE) y localidad (LOC). Estas tablas tienen en común el campo DEPT_NO, por el que se combinan las tablas:**

```
SQL> SELECT APELLIDO, OFICIO, EMP_NO, DNOMBRE, LOC FROM  EMPLE, DEPART
  2   WHERE EMPLE.DEPT_NO = DEPART.DEPT_NO;

    APELLIDO         OFICIO        EMP_NO      DNOMBRE            LOC
    ---------        ----------    ------      -------------      ----------
    SANCHEZ          EMPLEADO        7369      INVESTIGACION      MADRID
    ARROYO           VENDEDOR        7499      VENTAS             BARCELONA
    SALA             VENDEDOR        7521      VENTAS             BARCELONA
    JIMENEZ          DIRECTOR        7566      INVESTIGACION      MADRID
    MARTIN           VENDEDOR        7654      VENTAS             BARCELONA
    NEGRO            DIRECTOR        7698      VENTAS             BARCELONA
    CEREZO           DIRECTOR        7782      CONTABILIDAD       SEVILLA
    GIL              ANALISTA        7788      INVESTIGACION      MADRID
    REY              PRESIDENTE      7839      CONTABILIDAD       SEVILLA
    TOVAR            VENDEDOR        7844      VENTAS             BARCELONA
    ALONSO           EMPLEADO        7876      INVESTIGACION      MADRID
```

(Continúa)

(Continuación)

```
     APELLIDO           OFICIO         EMP_NO      DNOMBRE            LOC
     ---------          ----------     ------      -------------      ----------
     JIMENO             EMPLEADO         7900      VENTAS             BARCELONA
     FERNANDEZ          ANALISTA         7902      INVESTIGACION      MADRID
     MUÑOZ              EMPLEADO         7934      CONTABILIDAD       SEVILLA

14 filas seleccionadas.
```

Todos los empleados con un número de departamento 10 serán emparejados con los datos del departamento 10; los empleados con un número de departamento 20 se emparejarán con los datos del departamento 20 y, así, sucesivamente. Si se omite la cláusula WHERE, resultará un PRODUCTO CARTESIANO donde se emparejaría cada fila de una tabla con todas las filas de la otra tabla. En este caso: 14 filas de la tabla EMPLE por 4 filas de la tabla DEPART = 56 filas.

– Veamos un ejemplo en el que se combinan las siguientes 3 tablas.

  ALUMNOS: Contiene los datos de los alumnos. La descripción es ésta:

```
SQL> DESC ALUMNOS
     Nombre                                   ¿Nulo?        Tipo
     -------------------------------          --------      ------------
     DNI                                      NOT NULL      VARCHAR2(10)
     APENOM                                                 VARCHAR2(30)
     DIREC                                                  VARCHAR2(30)
     POBLA                                                  VARCHAR2(15)
     TELEF                                                  VARCHAR2(10)
```

  ASIGNATURAS: Contiene los nombres de asignaturas con sus códigos correspondientes. La descripción es la siguiente:

```
SQL> DESC ASIGNATURAS
     Nombre                                   ¿Nulo?        Tipo
     -------------------------------          --------      --------
     COD                                      NOT NULL      NUMBER(2)
     NOMBRE                                                 VARCHAR2(25)
```

  NOTAS: Contiene las notas de cada alumno en cada asignatura. Se relaciona con la tabla ALUMNOS por la columna DNI y con la tabla ASIGNATURAS por la columna COD. La descripción es:

```
SQL> DESC NOTAS
     Nombre                                   ¿Nulo?        Tipo
     -------------------------------          --------      --------
     DNI                                      NOT NULL      VARCHAR2(10)
     COD                                      NOT NULL      NUMBER(2)
     NOTA                                                   NUMBER(2)
```

– Realiza una consulta para obtener el nombre de alumno, su asignatura y su nota:

```
SQL> SELECT APENOM, NOMBRE, NOTA FROM ALUMNOS, ASIGNATURAS, NOTAS
  2   WHERE ALUMNOS.DNI=NOTAS.DNI AND NOTAS.COD=ASIGNATURAS.COD;
```

(Continúa)

(Continuación)

```
APENOM                          NOMBRE                       NOTA
------------------------        -----------------------    ------
Alcalde García, Elena           Prog. Leng. Estr.               6
Alcalde García, Elena           Sist. Informáticos              5
Alcalde García, Elena           Análisis                        6
Díaz Fernández, María           FOL                             8
Cerrato Vela, Luis              FOL                             6
Díaz Fernández, María           RET                             7
Cerrato Vela, Luis              RET                             8
Díaz Fernández, María           Entornos Gráficos               8
Cerrato Vela, Luis              Entornos Gráficos               4
Díaz Fernández, María           Aplic. Entornos 4ªGen           9
Cerrato Vela, Luis              Aplic. Entornos 4ªGen           5

11 filas seleccionadas.
```

– Obtén los nombres de alumnos matriculados en 'FOL':

```
SQL> SELECT APENOM FROM ALUMNOS, ASIGNATURAS, NOTAS WHERE
  2  ALUMNOS.DNI=NOTAS.DNI AND NOTAS.COD=ASIGNATURAS.COD
  3  AND NOMBRE='FOL';

     APENOM
     ----------------------
     Cerrato Vela, Luis
     Díaz Fernández, María
```

Actividades propuestas



6 **Visualiza los nombres de alumnos que tengan una nota entre 7 y 8 e la asignatura de "FOL".**

Visualiza los nombres de asignaturas que no tengan suspensos.

Conceptos básicos



Las Figuras 3.10 y 3.11 muestran un resumen de la sentencia SELECT vista hasta el momento.

CONSULTAS SIMPLES CON SELECT

SELECCIÓN DE COLUMNAS

```
SELECT * FROM EMPLE;
```

Selección: σ(EMPLE)

```
SELECT APELLIDO, SALARIO FROM EMPLE;
```

Proyección: $\Pi_{APELLIDO, SALARIO}$(EMPLE)

SELECCIÓN DE FILAS: WHERE

```
SELECT * FROM EMPLE WHERE DEPT_NO=20 OR SALARIO >1000;
```

Selección: $\sigma_{DEPTNO=20 \text{ OR } SALARIO>1000}$(EMPLE)

Condiciones básicas de búsqueda:

```
SELECT APELLIDO FROM EMPLE WHERE SALARIO BETWEEN 1000 AND 2000;
SELECT APELLIDO FROM EMPLE WHERE DEPT_NO IN(10,30);
SELECT APELLIDO FROM EMPLE WHERE APELLIDO LIKE '_R%';
SELECT APELLIDO FROM EMPLE WHERE COMISION IS NULL;
```

Condiciones de búsqueda complejas: OR, AND, NOT

Filas duplicadas- **DISTINCT:**

```
SELECT DISTINCT DEPT_NO FROM EMPLE;
```

Figura 3.10. *Consultas sencillas con SELECT.*

Ordenación de los resultados de una consulta: ORDER BY

Ascendente: **ASC**, opción por defecto. Descendente: **DESC**.

```
SELECT APELLIDO, SALARIO FROM EMPLE ORDER BY DEPT_NO DESC, APELLIDO;

SELECT APELLIDO, SALARIO+SALARIO*0.2  FROM EMPLE ORDER BY 2 DESC;
```

CONSULTAS MULTITABLA

DEPART: (DEPT_NO, DNOMBRE, LOC)
EMPLE: (EMP_NO, APELLIDO, SALARIO, COMISION, DEPT_NO, DIR)

PRODUCTO CARTESIANO

```
SELECT * FROM DEPART, EMPLE;
```

Producto cartesiano: DEPART X EMPLE

COMBINACIÓN INTERNA

```
SELECT * FROM DEPART, EMPLE WHERE DEPART.DEPT_NO = EMPLE.DEPT_NO;
```

Combinación: DEPART * $EMPLE_{(DEPART.DEPT_NO=EMPLE.DEPT_NO)}$

```
SELECT * FROM DEPART D, EMPLE E WHERE D.DEPT_NO = E.DEPT_NO
```

Alias de tablas

Figura 3.11. *Combinación de tablas.*





Actividades complementarias



Tablas EMPLE y DEPART

1. Selecciona el apellido, el oficio y la localidad de los departamentos de aquellos empleados cuyo oficio sea "ANALISTA".

2. Obtén los datos de los empleados cuyo director (columna DIR de la tabla EMPLE) sea "CEREZO".

3. Obtén los datos de los empleados del departamento de "VENTAS".

4. Obtén los datos de los departamentos que NO tengan empleados.

5. Obtén los datos de los departamentos que tengan empleados.

6. Obtén el apellido y el salario de los empleados que superen todos los salarios de los empleados del departamento 20.

Tabla LIBRERIA

7. Visualiza el tema, estante y ejemplares de las filas de LIBRERIA con ejemplares comprendidos entre 8 y 15.

8. Visualiza las columnas TEMA, ESTANTE y EJEMPLARES de las filas cuyo ESTANTE no esté comprendido entre la "B" y la "D".

9. Visualiza con una sola orden SELECT todos los temas de LIBRERIA cuyo número de ejemplares sea inferior a los que hay en "MEDICINA".

10. Visualiza los temas de LIBRERIA cuyo número de ejemplares no esté entre 15 y 20, ambos incluidos.

Tablas ALUMNOS, ASIGNATURAS y NOTAS

11. Visualiza todas las asignaturas que contengan tres letras "o" en su interior y tengan alumnos matriculados de "Madrid".

12. Visualiza los nombres de alumnos de "Madrid" que tengan alguna asignatura suspendida.

13. Muestra los nombres de alumnos que tengan la misma nota que tiene "Díaz Fernández, María" en "FOL" en alguna asignatura.

14. Obtén los datos de las asignaturas que no tengan alumnos.

15. Obtén el nombre y apellido de los alumnos que tengan nota en la asignatura con código 1.

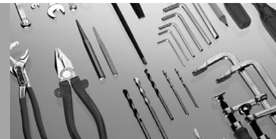
16. Obtén el nombre y apellido de los alumnos que no tengan nota en la asignatura con código 1.

Funciones

4

En esta unidad aprenderás a:

1. **Identificar las distintas funciones que se pueden usar con la cláusula SELECT.**
2. **Aplicar las diferentes funciones para obtener información de expresiones o de las columnas de las tablas.**





4.1 Introducción

Las *funciones* se usan dentro de expresiones y actúan con los valores de las columnas, variables o constantes. Generalmente producen dos tipos diferentes de resultados: unas producen una modificación de la información original (por ejemplo, poner en minúscula una cadena que está en mayúscula); el resultado de otras indica alguna cosa sobre la información (por ejemplo, el número de caracteres que tiene una cadena). Se utilizan en: cláusulas SELECT, cláusulas WHERE y cláusulas ORDER BY.

Es posible el anidamiento de funciones. Existen cinco tipos de funciones: *aritméticas*, *de cadenas de caracteres*, *de manejo de fechas*, *de conversión* y *otras funciones*.

4.2 Funciones aritméticas

Las **funciones aritméticas** trabajan con datos de tipo numérico NUMBER. Este tipo incluye los dígitos de 0 a 9, el punto decimal y el signo menos, si es necesario. Los literales numéricos no se encierran entre comillas. Ejemplo: –123.32.

Estas funciones trabajan con tres clases de números: *valores simples*, *grupos de valores* y *listas de valores*. Algunas modifican los valores sobre los que actúan; otras informan de algo sobre los valores. Podemos dividir las funciones aritméticas en tres grupos:

- *Funciones de valores simples.*
- *Funciones de grupos de valores.*
- *Funciones de listas.*

Al describir los formatos de las funciones utilizaremos los corchetes ([ ]) para indicar que lo que va encerrado es opcional.

A. Funciones de valores simples

Las **funciones de valores simples** son funciones sencillas que trabajan con valores simples. Un valor simple es: un número (como 6522,90), una variable o una columna de una tabla.

Para probar algunas de estas funciones usaremos la tabla DUAL, cuya descripción es la siguiente:

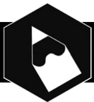
```
SQL> DESC DUAL;

       Nombre           ¿Nulo?     Tipo
       -----------     -------    -----------
       DUMMY                      VARCHAR2(1)
```

Las funciones de valores simples se muestran en la Tabla 4.1.

Función	Propósito
ABS(n)	Devuelve el valor absoluto de 'n'. El valor absoluto es siempre un número positivo.
CEIL(n)	Obtiene el valor entero inmediatamente superior o igual a 'n'.
FLOOR(n)	Es lo opuesto a CEIL, devuelve el valor entero inmediatamente inferior o igual a 'n'.
MOD(m, n)	Devuelve el resto resultante de dividir 'm' entre 'n'.
NVL(valor, expresión)	Esta función se utiliza para sustituir un valor nulo por otro valor. Si 'valor' es NULL, es sustituido por la 'expresión'; si no lo es, la función devuelve 'valor'. NVL se puede usar con cualquier tipo de datos: numéricos, carácter, tipo fecha, pero 'valor' y 'expresión' deben ser del mismo tipo, aunque admiten tipos diferentes.
POWER(m, exponente)	Calcula la potencia de un número. Devuelve el valor de 'm' elevado a un 'exponente'.
ROUND(número [,m])	Devuelve el valor de 'número' redondeado a 'm' decimales. Si 'm' es negativo, el redondeo de dígitos se lleva a cabo a la izquierda del punto decimal. Si se omite 'm', devuelve 'número' con 0 decimales y redondeado.
SIGN(valor)	Esta función indica el signo del 'valor'. Si 'valor' es menor que 0, la función devuelve _1; y si 'valor' es mayor que 0, la función devuelve 1.
SQRT(n)	Devuelve la raíz cuadrada de 'n', El valor de 'n' no puede ser negativo.
TRUNC(número, [m])	Trunca los números para que tengan un cierto número de dígitos de precisión. Devuelve 'número' truncado a 'm' decimales; 'm' puede ser negativo: si lo es, trunca por la izquierda del punto decimal. Si se omite 'm' devuelve 'número' con 0 decimales.

Tabla 4.1. *Funciones de valores simples.*



Caso práctico

1 **ABS(n).** Obtén el valor absoluto del SALARIO - 10000 para todas las filas de la tabla EMPLE:

```
SQL> SELECT APELLIDO,  SALARIO, ABS(SALARIO-10000) FROM EMPLE;
```

CEIL(n). Prueba con números positivos y negativos:

```
SQL> SELECT CEIL(20.3), CEIL(16), CEIL(-20.3), CEIL(-16)  FROM DUAL;

    CEIL(20.3)       CEIL(16)    CEIL(-20.3)      CEIL(-16)
    ----------      ---------    -----------      ---------
            21             16            -20            -16
```

(Continúa)





(Continuación)

FLOOR(n). Prueba con números positivos y negativos:

```
SQL> SELECT FLOOR(20.3), FLOOR (16), FLOOR(-20.3), FLOOR(-16)  FROM DUAL;

      FLOOR(20.3)          FLOOR(16)      FLOOR(-20.3)        FLOOR(-16)
      -----------        ----------      ------------        ----------
               20                 16               -21               -16
```

MOD(m, n). Prueba con números positivos y negativos:

```
SQL> SELECT MOD(11,4), MOD(10,-15), MOD(-10,-3), MOD(10.4,4.5) FROM DUAL;

        MOD(11,4)       MOD(10,-15)       MOD(-10,-3)     MOD(10.4,4.5)
        ---------       -----------       -----------     -------------
                3                10                -1               1,4
```

NVL(valor, expresión). A partir de la tabla EMPLE obtenemos el SALARIO, la COMISION y la suma de ambos:

```
SQL> SELECT SALARIO, COMISION, SALARIO + COMISION  FROM EMPLE;

          SALARIO         COMISION       SALARIO+COMISION
       ----------        ----------      ----------------
             1040
             1500               390                  1890
             1625               650                  2275
             2900
             1600              1020                  2620
             3005
             2885
             3000
             4100
             1350                 0                  1350
             1430
             1335
             3000
             1690

14 filas seleccionadas.
```

En este ejemplo, se obtiene la suma del SALARIO y la COMISION. Si la comisión es nula, la columna SALARIO + COMISION da como resultado un valor nulo (no se visualiza nada), debido a que los valores nulos en las expresiones siempre darán como resultado un valor nulo.

A partir de la tabla EMPLE obtenemos el SALARIO, la COMISION y la suma de ambos, pero aplicando la función NVL a la comisión. Si es nula sustituimos su valor por 0:

```
SQL> SELECT SALARIO,COMISION, SALARIO + NVL(COMISION,0) FROM EMPLE;
```

(Continúa)

(Continuación)

```
   SALARIO        COMISION      SALARIO+NVL(COMISION,0)
----------      ----------      -----------------------
      1040                                         1040
      1500             390                         1890
      1625             650                         2275
      2900                                         2900
      1600            1020                         2620
      3005                                         3005
      2885                                         2885
      3000                                         3000
      4100                                         4100
      1350               0                         1350
      1430                                         1430
      1335                                         1335
      3000                                         3000
      1690                                         1690

14 filas seleccionadas.
```

En este otro ejemplo, al aplicar a la comisión la función NVL en la columna SALARIO + COMISION, no resultan valores nulos.

POWER(m, exponente). Prueba con números positivos y negativos:

```
SQL> SELECT POWER(2,4),POWER(2,-4),POWER(3.5, 2.4), POWER(4.5, 2)  FROM DUAL;

 POWER(2,4)     POWER(2,-4)      POWER(3.5,2.4)      POWER(4.5,2)
 ----------     -----------      --------------      ------------
         16           ,0625          20,2191692             20,25
```

ROUND(número [,m]). Prueba con redondeo positivo:

```
SQL> SELECT ROUND(1.56,1), ROUND(1.56), ROUND(1.2234,2), ROUND(1.2676, 3)  FROM DUAL;

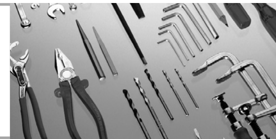
 ROUND(1.56,1)    ROUND(1.56)     ROUND(1.2234,2)    ROUND(1.2676,3)
 -------------    -----------     ---------------    ---------------
           1,6              2                1,22              1,268
```

Prueba con redondeo negativo:

```
SQL> SELECT ROUND(145.5, -1), ROUND(145.5, -2), ROUND(145.5, -3), ROUND(141,-1),
ROUND(145,-1) FROM DUAL;

 ROUND(145.5,-1)  ROUND(145.5,-2)  ROUND(145.5,-3) ROUND(141,-1)  ROUND(145,-1)
 ---------------  ---------------  --------------- -------------  -------------
             150              100                0           140            150
```

(Continúa)





(Continuación)

SIGN(valor). Prueba con números positivos y negativos:

```
SQL> SELECT SIGN(-10), SIGN(10) FROM DUAL;

     SIGN(-10)             SIGN(10)
    ----------           ----------
            -1                    1
```

SQRT(n). `SQL> SELECT SQRT(25), SQRT(25.6) FROM DUAL;`

```
      SQRT(25)           SQRT(25.6)
      --------           ----------
             5           5,05964426
```

TRUNC(número, [m]).

Con truncamiento positivo:

```
SQL> SELECT TRUNC(1.5634, 1), TRUNC(1.1684, 2), TRUNC(1.662) FROM DUAL;

  TRUNC(1.5634,1)        TRUNC(1.1684,2)          TRUNC(1.662)
  ---------------        ---------------          ------------
              1,5                   1,16                     1
```

Con truncamiento negativo:

```
SQL> SELECT TRUNC(187.98,-1), TRUNC(187.98,-2), TRUNC(187.98,-3) FROM DUAL;

 TRUNC(187.98,-1)       TRUNC(187.98,-2)       TRUNC(187.98,-3)
 ----------------       ----------------       ----------------
              180                    100                      0
```

Actividades propuestas



1 ¿Cuál sería la salida de ejecutar estas funciones?

ABS(146)=	ABS(-30)=	POWER(3,-1)=	ROUND(33.67)=
CEIL(2)=	CEIL(1.3)=	ROUND(-33.67,2)=	ROUND(-33.67,-2)=
CEIL(-2.3)=	CEIL(-2)=	ROUND(-33.27,1)=	ROUND(-33.27,-1)=
FLOOR(-2)=	FLOOR(-2.3)=	TRUNC(67.232)=	TRUNC(67.232,-2)=
FLOOR(2)=	FLOOR(1.3)=	TRUNC(67.232,2)=	TRUNC(67.58,-1)=
MOD(22,23)=	MOD(10,3)=	TRUNC(67.58,1)=	
POWER(10,0)=	POWER(3,2)=		

B. Funciones de grupos de valores

Hasta ahora nos hemos ocupado de funciones que operan con valores simples; no obstante, hay otras funciones estadísticas, como SUM, AVG y COUNT, que actúan sobre un grupo de filas para obtener un valor. Estas funciones permiten obtener la edad media de un grupo de alumnos, el alumno más joven, el más viejo, el número total de miembros de un grupo, etcétera. Los valores nulos son ignorados por las funciones de grupos de valores y los cálculos se realizan sin contar con ellos. Estas funciones se muestran en la Tabla 4.2.

Función	Propósito
AVG(n)	Calcula el valor medio de 'n' ignorando los valores nulos.
COUNT (* | expresión)	Cuenta el número de veces que la expresión evalúa algún dato con valor no nulo. La opción '*' cuenta todas las filas seleccionadas.
MAX(expresión)	Calcula el máximo valor de la 'expresión'.
MIN(expresión)	Calcula el mínimo valor de la 'expresión'.
SUM(expresión)	Obtiene la suma de valores de la 'expresión' distintos de nulos.
VARIANCE (expresión)	Obtiene la varianza de los valores de 'expresión' distintos de nulos.

Tabla 4.2. *Funciones de grupos de valores.*

Caso práctico

2 **AVG(n).** Cálculo del salario medio de los empleados del departamento 10 de la tabla EMPLE:

```
SQL> SELECT AVG(SALARIO) FROM EMPLE WHERE DEPT_NO=10;

      AVG(SALARIO)
      ------------
        2891,66667
```

COUNT (* | expresión). Cálculo del número de filas de la tabla EMPLE:

```
SQL> SELECT COUNT(*) FROM EMPLE;

          COUNT(*)
          --------
                14
```

Cálculo del número de filas de la tabla EMPLE donde la COMISION no es nula:

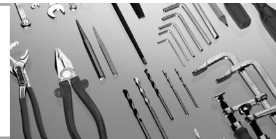
```
SQL> SELECT COUNT(COMISION) FROM EMPLE;

     COUNT(COMISION)
     ---------------
                   4
```

MAX(expresión). Cálculo del máximo salario de la tabla EMPLE:

```
SQL> SELECT MAX(SALARIO) FROM EMPLE;
```

(Continúa)





(Continuación)

```
MAX(SALARIO)
------------
        4100
```

Obtén el apellido máximo (alfabéticamente) de la tabla EMPLE:

```
SQL> SELECT MAX(APELLIDO) FROM EMPLE;

MAX(APELLI
----------
TOVAR
```

Obtén el apellido del empleado que tiene mayor salario:

```
SQL> SELECT APELLIDO, SALARIO FROM EMPLE WHERE SALARIO=(SELECT MAX(SALARIO) FROM EMPLE);

APELLIDO        SALARIO
----------   ----------
REY                4100
```

Se necesita una subconsulta para calcular el máximo salario y compararlo después con el salario de cada uno de los empleados de la tabla EMPLE.

MIN(expresión). Obtén el mínimo salario de la tabla EMPLE:

```
SQL> SELECT MIN(SALARIO) FROM EMPLE;

MIN(SALARIO)
------------
        1040
```

Obtén el apellido del empleado que tiene mínimo salario:

```
SQL> SELECT APELLIDO, SALARIO FROM EMPLE WHERE SALARIO=(SELECT MIN(SALARIO) FROM EMPLE);

APELLIDO        SALARIO
----------   ----------
SANCHEZ            1040
```

En primer lugar, se selecciona el salario mínimo y, a continuación, se compara con los salarios de la tabla EMPLE para obtener el apellido que tiene ese salario mínimo.

SUM(expresión). Consigue la suma de todos los salarios de la tabla EMPLE:

```
SQL> SELECT SUM(SALARIO) FROM EMPLE;

SUM(SALARIO)
------------
       30460
```

VARIANCE(expresión). Obtén la varianza de todos los salarios de la tabla EMPLE:

```
SQL> SELECT VARIANCE(SALARIO) FROM EMPLE;

VARIANCE(SALARIO)
-----------------
       872226,374
```

DISTINCT en funciones de grupo

En todas las funciones de grupo, al indicar los argumentos se pueden emplear las **cláusulas DISTINCT** y **ALL,** aunque no se suelen utilizar en las funciones AVG, SUM, MAX ni MIN, pero sí es más normal su uso en COUNT.

Recordemos que DISTINCT realiza una selección de filas cuyos valores en la columna especificada no están duplicados. La cláusula ALL recoge todas las filas aunque sus valores estén duplicados.

El formato de COUNT incluyendo DISTINCT y ALL es éste:

```
COUNT ( * | [DISTINCT | ALL]  expresión)
```

Si COUNT recibe como argumento una expresión o columna, ésta podrá ir precedida de las cláusulas ALL o DISTINCT.



Caso práctico

3 **Calcula el número de oficios que hay en la tabla EMPLE:**

```
SQL> SELECT  COUNT(OFICIO) "OFICIOS"  FROM EMPLE;

  OFICIOS
 --------
       14
```

Esta consulta cuenta todos los oficios de la tabla EMPLE que no sean nulos, estén repetidos o no. Si queremos contar los distintos oficios que hay en la tabla EMPLE, tendríamos que incluir DISTINCT en la función de grupo: `SQL> SELECT COUNT(DISTINCT OFICIO) "OFICIOS"  FROM EMPLE;`

```
  OFICIOS
 --------
        5
```

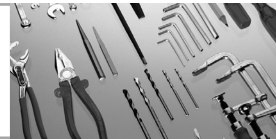
DISTINCT obliga a COUNT a contar sólo el número de oficios distintos.



Actividades propuestas

2 **A partir de la tabla EMPLE, visualiza cuántos apellidos empiezan por la letra 'A'.**

Obtén el apellido o apellidos de empleados que empiecen por la letra 'A' y que tengan máximo salario (de los que empiezan por la letra 'A').





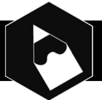
C. Funciones de listas

Las **funciones de listas** trabajan sobre un grupo de columnas dentro de una misma fila. Comparan los valores de cada una de las columnas en el interior de una fila para obtener el mayor o el menor valor de la lista. Las funciones de listas se muestran en la Tabla 4.3.

Función	Propósito
GREATEST (valor1, valor2, ...)	Obtiene el mayor valor de la lista.
LEAST (valor1, valor2, ...)	Obtiene el menor valor de la lista.

Tabla 4.3. *Funciones de listas.*

Caso práctico



4 Sea la tabla NOTAS_ALUMNOS:

```
SQL> DESC NOTAS_ALUMNOS;

       Nombre                 ¿Nulo?           Tipo
       ----------------       ----------       ------------
       NOMBRE_ALUMNO          NOT NULL         VARCHAR2(25)
       NOTA1                                   NUMBER(2)
       NOTA2                                   NUMBER(2)
       NOTA3                                   NUMBER(2)
```

Obtén por cada alumno la mayor nota y la menor nota de las tres que tiene:

```
SQL>SELECT NOMBRE_ALUMNO,GREATEST(NOTA1,NOTA2,NOTA3)"MAYOR",LEAST(NOTA1,NOTA2,
NOTA3)"MENOR"  FROM NOTAS_ALUMNOS;

    NOMBRE_ALUMNO                          MAYOR             MENOR
    -------------------------         ----------        ----------
    Alcalde García, M. Luisa                   5                 5
    Benito Martín, Luis                        8                 6
    Casas Martínez, Manuel                     7                 5
    Corregidor Sánchez, Ana                    9                 6
    Díaz Sánchez, Maria
```

La última fila tiene un resultado de NULL, debido a que GREATEST y LEAST no pueden comparar un valor con otro valor nulo. Estas funciones se pueden usar también con columnas de caracteres. Ejemplos: Obtén el mayor nombre alfabético de la lista:

```
SQL> SELECT GREATEST('BENITO','JORGE','ANDRES','ISABEL') FROM DUAL;

         GREAT
         -----
         JORGE
```

4.3 Funciones de cadenas de caracteres

Las **funciones de cadenas de caracteres** trabajan con datos de tipo CHAR o VARCHAR2. Estos datos incluyen cualquier carácter alfanumérico: letras, números y caracteres especiales. Los literales se deben encerrar entre comillas simples. Ejemplo de una cadena de caracteres: 'El Quijote'.

Las funciones de cadenas permiten manipular cadenas de letras u otros caracteres. Estas funciones pueden calcular el número de caracteres de una cadena, convertir una cadena a mayúsculas o a minúsculas, suprimir o añadir caracteres a la izquierda o a la derecha, etcétera.

A. Funciones que devuelven valores carácter

Estas funciones devuelven un carácter o un conjunto de caracteres; son un ejemplo las funciones que devuelven una cadena en mayúscula o una cadena en minúscula, o las que obtienen parte de una cadena. Se muestran en la Tabla 4.4.

Función	Propósito
CHR (n)	Devuelve el carácter cuyo valor en binario es equivalente a 'n'.
CONCAT (cad1, cad2)	Devuelve 'cad1' concatenada con 'cad2'. Es equivalente al operador ||.
LOWER (cad)	Devuelve la cadena 'cad' con todas sus letras convertidas a minúsculas.
UPPER (cad)	Devuelve la cadena 'cad' con todas sus letras convertidas a mayúsculas.
INITCAP (cad)	Convierte la cadena 'cad' a tipo título, la primera letra de cada palabra de 'cad' a mayúsculas y el resto, a minúsculas.
LPAD (cad1, n [, cad2])	Esta función añade caracteres a la izquierda de 'cad1' hasta que alcance una cierta longitud 'n'. Devuelve 'cad1' con longitud 'n' y ajustado a la derecha; 'cad2' es la cadena con la que se rellena por la izquierda; cad1 puede ser una columna de una tabla o cualquier literal. Si 'cad2' se suprime, asume como carácter de relleno el blanco.
RPAD (cad1, n [, cad2])	Añade caracteres a la derecha de 'cad1' hasta que alcance una cierta longitud 'n'. Devuelve 'cad1' con longitud 'n' y ajustado a la izquierda; 'cad2' es la cadena con la que se rellena por la derecha; 'cad1' puede ser una columna de una tabla o cualquier literal. Si 'cad2' se suprime, asume como carácter de relleno el blanco.
LTRIM (cad [, set])	Suprime un conjunto de caracteres a la izquierda de la cadena 'cad'; 'set' es el conjunto de caracteres a suprimir. Devuelve 'cad' con el grupo de caracteres 'set' omitidos por la izquierda. Si el segundo parámetro se omite, devuelve la misma cadena. Por defecto, si la cadena contiene blancos a la izquierda y se omite el segundo parámetro, la función devuelve la cadena sin blancos a la izquierda.
RTRIM (cad [, set])	Suprime un conjunto de caracteres a la derecha de la cadena 'cad'. Devuelve 'cad' con el grupo de caracteres 'set' omitidos por la derecha. Si se omite 'set', devuelve 'cad' tal como está. Por defecto, si la cadena contiene blancos a la derecha y se omite el segundo parámetro, la función devuelve la cadena sin blancos a la derecha.
REPLACE (cad, cadena_búsqueda [,cadena_sustitución])	Sustituye un carácter o varios caracteres de una cadena con 0 o más caracteres. Devuelve 'cad' con cada ocurrencia de 'cadena_búsqueda' sustituida por 'cadena_sustitución'.
SUBSTR (cad, m [,n])	Obtiene parte de una cadena. Devuelve la subcadena de 'cad', que abarca desde la posición indicada en 'm' hasta tantos caracteres como indique el número 'n'. Si se omite 'n', devuelve la cadena desde la posición especificada por 'm'. El valor de 'n' no puede ser inferior a 1. El valor de 'm' puede ser negativo; en ese caso, devuelve la cadena empezando por su final, y avanzando de derecha a izquierda.
TRANSLATE (cad1, cad2, cad3)	Convierte caracteres de una cadena en caracteres diferentes, según un plan de sustitución marcado por el usuario. Devuelve 'cad1' con los caracteres encontrados en 'cad2' y sustituidos por los caracteres de 'cad3'. Cualquier carácter que no esté en la cadena 'cad2' permanece como estaba.

Tabla 4.4. *Funciones que devuelven factores carácter.*





Caso práctico



5 **CHR(n).** Devuelve las letras cuyo valor ASCII es 75 y 65:

```
SQL> SELECT  CHR(75), CHR(65) FROM DUAL;

  C     C
  -     -
  K     A
```

CONCAT(cad1, cad2). Obtén el apellido de los empleados de la tabla EMPLE de la siguiente manera: El apellido es: APELLIDO.

Para ello necesitamos concatenar la cadena 'El apellido es: ' con la columna APELLIDO de la tabla EMPLE: `SQL> SELECT CONCAT('El apellido es: ', APELLIDO) FROM EMPLE;`

```
CONCAT('ELAPELLIDOES:',APE
--------------------------
El apellido es: SANCHEZ
El apellido es: ARROYO
El apellido es: SALA
El apellido es: JIMENEZ
El apellido es: MARTIN
El apellido es: NEGRO
El apellido es: CEREZO
El apellido es: GIL
El apellido es: REY
El apellido es: TOVAR
El apellido es: ALONSO
El apellido es: JIMENO
El apellido es: FERNANDEZ
El apellido es: MUÑOZ

14 filas seleccionadas.
```

LOWER(cad), UPPER(cad) e INITCAP(cad). Visualiza en mayúsculas, minúsculas y tipo título la cadena: 'oRACle Y sqL':

```
SQL> SELECT LOWER('oRACle Y sqL') "minuscula", UPPER('oRACle Y sqL') "MAYUSCULA",
INITCAP('oRACle Y sqL') "Tipo Titulo" FROM DUAL;

    minuscula           MAYUSCULA           Tipo Titulo
    ------------        ------------        ------------
    oracle y sql        ORACLE Y SQL        Oracle Y Sql
```

LPAD(cad1, n [, cad2]) y RPAD(cad1, n [, cad2]). Para cada fila de la tabla NOTAS_ALUMNOS, obtenemos en una columna el nombre del alumno con una longitud de 30 caracteres y rellenando por la izquierda con puntos y en otra columna lo mismo pero rellenando por la derecha: `SQL> SELECT LPAD(NOMBRE_ALUMNO,30,'.') "IZQUIERDA", RPAD(NOMBRE_ALUMNO,30,'.') "DERECHA" FROM NOTAS_ALUMNOS;`

```
  IZQUIERDA                         DERECHA
  ------------------------------    ------------------------------
  ......Alcalde García, M. Luisa    Alcalde García, M. Luisa......
```

(Continúa)

(Continuación)

```
...........Benito Martín, Luis       Benito Martín, Luis...........
........Casas Martínez, Manuel       Casas Martínez, Manuel........
.......Corregidor Sánchez, Ana       Corregidor Sánchez, Ana.......
...........Díaz Sánchez, Maria       Díaz Sánchez, Maria...........
```

LTRIM(cad [, set]) y RTRIM(cad [, set]). Usamos las funciones LTRIM y RTRIM sin el segundo parámetro y con una cadena con blancos a la izquierda y a la derecha que, por defecto, eliminará:

```
SQL> SELECT LTRIM('       hola') || RTRIM('     adios  ')||'*' FROM DUAL;

      LTRIM('HOLA')||R
      ----------------
      hola     adios*
```

A partir de la tabla MISTEXTOS: `SQL> SELECT * FROM MISTEXTOS;`

```
TITULO                           AUTOR              EDITORIAL          PAGINA
-------------------------------  -----------------  ---------------  ---------
METODOLOGÍA DE LA PROGRAMACIÓN.  ALCALDE GARCÍA     MCGRAWHILL             140
"INFORMÁTICA BÁSICA."            GARCÍA GARCERÁN    PARANINFO              130
SISTEMAS OPERATIVOS              GARCÍA ESTRUCH     OBSBORNE               300
SISTEMAS DIGITALES.              RUÍZ LOPEZ         PRENTICE HALL          190
"MANUAL DE C."                   RUÍZ LOPEZ         MCGRAWHILL             340
```

Quitamos los caracteres punto y comilla de la derecha de la columna TITULO:

```
SQL> SELECT RTRIM (TITULO, '."' ) FROM MISTEXTOS;

  RTRIM(TITULO,'."')
  --------------------------------
  METODOLOGÍA DE LA PROGRAMACIÓN
  "INFORMÁTICA BÁSICA
  SISTEMAS OPERATIVOS
  SISTEMAS DIGITALES
  "MANUAL DE C
```

Quitamos las comillas de la izquierda de la columna TITULO:

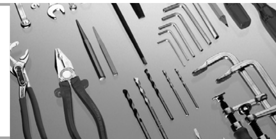
```
SQL> SELECT LTRIM (TITULO, '"' ) FROM MISTEXTOS;

  LTRIM(TITULO,'"')
  --------------------------------
  METODOLOGÍA DE LA PROGRAMACIÓN.
  INFORMÁTICA BÁSICA."
  SISTEMAS OPERATIVOS
  SISTEMAS DIGITALES.
  MANUAL DE C."
```

REPLACE(cad, cadena_búsqueda [,cadena_sustitución]). Sustituimos 'O' por 'AS' en la cadena 'BLANCO Y NEGRO':

```
SQL> SELECT  REPLACE('BLANCO Y NEGRO', 'O', 'AS')  FROM  DUAL;
```

(Continúa)





(Continuación)

```
REPLACE('BLANCOY
----------------
BLANCAS Y NEGRAS
```

Si no ponemos nada en la cadena_sustitución, se sustituiría la cadena_búsqueda por nada (NULL). En el siguiente ejemplo, sustituimos 'O' por nada:

```
SQL> SELECT  REPLACE('BLANCO Y NEGRO', 'O') FROM DUAL;

    REPLACE('BLA
    ------------
    BLANC Y NEGR
```

SUBSTR(cad, m [,n]). Partiendo de la cadena 'ABCDEF', obtenemos en una columna dos caracteres a partir de la tercera posición, en otra columna otros dos caracteres a partir de la tercera posición empezando por el final de la cadena y en una última columna la cadena a partir de su cuarta posición:

```
SQL>  SELECT  SUBSTR('ABCDEFG',3,2)  "s1",  SUBSTR  ('ABCDEFG',-3,2)  "s2",
SUBSTR('ABCDEFG',4) "s3" FROM DUAL;

    s1       s2       s3
    --       --       ----
    CD       EF       DEFG
```

Visualiza el apellido y su primera letra para los empleados del departamento 10 de la tabla EMPLE:

```
SQL> SELECT APELLIDO, SUBSTR(APELLIDO,1,1) FROM EMPLE WHERE DEPT_NO =10;

  APELLIDO        S
  ----------      -
  CEREZO          C
  REY             R
  MUÑOZ           M
```

TRANSLATE(cad1, cad2 , cad3). A partir de la cadena 'LOS PILARES DE LA TIERRA', sustituye la 'A' por 'a', la 'E' por 'e', la 'I' por 'i', la 'O' por 'o', y la 'U' por 'u':

```
SQL> SELECT TRANSLATE('LOS PILARES DE LA TIERRA','AEIOU','aeiou') FROM DUAL;

  TRANSLATE('LOSPILARESDEL
  ------------------------
  LoS PiLaReS De La TieRRa
```

En este ejemplo TRANSLATE sustituye la 'L' por 'l':

```
SQL> SELECT TRANSLATE('LOS PILARES DE LA TIERRA','LAEIOU','l') FROM DUAL;

  TRANSLATE('LOSP
  ---------------
  lS PlRS D l TRR
```

Ahora la cadena "cad2" está formada por 'LAEIOU', pero en la cadena "cad3" sólo existe 'l', lo que hace que desaparezcan las vocales en la cadena resultante; ya que sólo la 'L' se reemplaza por 'l'; el resto, 'AEIOU', se reemplaza por nada.

B. Funciones que devuelven valores numéricos

Estas funciones devuelven valores numéricos, como el número de caracteres que tiene una cadena o la posición en la que se encuentra un determinado carácter en una cadena. Se trata de las siguientes mostradas en la Tabla 4.5.

Función	Propósito
ASCII(cad)	Devuelve el valor ASCII de la primera letra de la cadena 'cad'.
INSTR(cad1,cad2 [,comienzo [,m]])	Esta función busca un conjunto de caracteres en una cadena. Devuelve la posición de la 'm_ésima' ocurrencia de 'cad2' en 'cad1', empezando la búsqueda en la posición 'comienzo'. Por omisión, empieza buscando en la posición 1.
LENGTH(cad)	Devuelve el número de caracteres de 'cad'.

Tabla 4.5. *Funciones que devuelven valores numéricos.*



Caso práctico

6 **ASCII(cad).** Obtén el valor ASCII de 'A': `SQL> SELECT ASCII ('A') FROM DUAL;`

```
ASCII('A')
----------
        65
```

Si ponemos como argumento una cadena de caracteres, visualiza sólo el valor ASCII del primer carácter de la cadena.

INSTR(cad1, cad2 [,comienzo [,m]]). A partir de la cadena 'II VUELTA CICLISTA A TALAVERA' encuentra la segunda ocurrencia 'TA' desde la posición 3:

```
SQL> SELECT INSTR('II VUELTA CICLISTA A TALAVERA', 'TA', 3, 2) "EJEMPLO" FROM DUAL;

   EJEMPLO
----------
        17
```

Cuando «comienzo» es negativo (-1), se comienza la búsqueda en la posición final y se va de derecha a izquierda en la cadena. Devuelve la posición contando desde la izquierda, es decir, la primera A que encuentra desde la primera posición empezando por el final:

```
SQL> SELECT INSTR('II VUELTA CICLISTA A TALAVERA', 'A', -1) "EJEMPLO" FROM  DUAL;

   EJEMPLO
----------
        29
```

(Continúa)





(Continuación)

Si en el ejemplo anterior empleamos un –2, haría que la función comenzase desde la segunda posición empezando por el final; un –3 haría que se iniciase desde la tercera posición y, así, sucesivamente.

A partir de la tabla MISTEXTOS, encuentra la posición de la segunda ocurrencia de la letra 'A' en la columna AUTOR a partir del comienzo:

```
SQL> SELECT AUTOR, INSTR(AUTOR,'A',1,2) FROM MISTEXTOS;

  AUTOR                        INSTR(AUTOR,'A',1,2)
  ----------------------     --------------------
  ALCALDE GARCÍA                                  4
  GARCÍA GARCERÁN                                 6
  GARCÍA STRUCH                                   6
  RUÍZ LOPEZ                                      0
  RUÍZ LOPEZ                                      0
```

Si en la función INSTR, "cad2" es un conjunto de caracteres, entonces la función devuelve la posición donde comienza el primer carácter de ese conjunto.

LENGTH(cad). Calcula el número de caracteres de las columnas TITULO y AUTOR para todas las filas de la tabla MISTEXTOS:

```
SQL> SELECT TITULO, LENGTH(TITULO), AUTOR, LENGTH(AUTOR) FROM MISTEXTOS;

  TITULO                          LENGTH(TITULO)  AUTOR            LENGTH(AUTOR)
  ------------------------------- --------------  ---------------- -------------
  METODOLOGÍA DE LA PROGRAMACIÓN.             31  ALCALDE GARCÍA              14
  "INFORMÁTICA BÁSICA."                       21  GARCÍA GARCERÁN             15
  SISTEMAS OPERATIVOS                         19  GARCÍA STRUCH               13
  SISTEMAS DIGITALES.                         19  RUÍZ LOPEZ                  10
  "MANUAL DE C."                              14  RUÍZ LOPEZ                  10
```

4.4 Funciones para el manejo de fechas

Oracle puede almacenar datos de tipo fecha (DATE) y posee una interesante utilidad para formatear las fechas de cualquier manera que se pueda concebir. Tiene un formato por omisión: 'DD/MM/YY', pero con la función TO_CHAR es posible mostrar las fechas de cualquier modo. Los literales de fecha deben encerrarse entre comillas simples.

Ejemplo: '18/11/05'.

Recordemos que el tipo de datos DATE se almacena en un formato especial que incluye Siglo/Año/Mes/Día/Hora/Minutos/Segundos. Las funciones para el manejo de fechas se exponen en la Tabla 4.6.

Función	Propósito
SYSDATE	Devuelve la fecha del sistema.
ADD_MONTHS(fecha, n)	Devuelve la fecha 'fecha' incrementada en 'n' meses.
LAST_DAY(fecha)	Devuelve la fecha del último día del mes que contiene 'fecha'.
MONTHS_BETWEEN (fecha1, fecha2)	Devuelve la diferencia en meses entre las fechas 'fecha1' y 'fecha2'.
NEXT_DAY(fecha, cad)	Devuelve la fecha del primer día de la semana indicado por 'cad' después de la fecha indicada por 'fecha'. El día de la semana en 'cad' se indica con su nombre, es decir, lunes (*monday*), martes (*tuesday*), miércoles (*wednesday*), jueves (*thursday*), viernes (*friday*), sábado (*saturday*) o domingo (*sunday*).

Tabla 4.6. *Funciones para el manejo de fechas.*



Caso práctico

7 **SYSDATE.** Esta función devuelve la fecha del sistema. Por ejemplo:

```
SQL> SELECT SYSDATE FROM DUAL;

      SYSDATE
     --------
     03/08/05
```

ADD_MONTHS(fecha, n). A partir de la tabla EMPLE, suma doce meses a la fecha de alta para los empleados del departamento 10:

```
SQL> SELECT FECHA_ALT, ADD_MONTHS(FECHA_ALT, 12)  FROM EMPLE WHERE DEPT_NO=10;

     FECHA_AL            ADD_MONT
     --------            --------
     09/06/91            09/06/92
     17/11/91            17/11/92
     23/01/92            23/01/93
```

LAST_DAY(fecha). Obtén de la tabla EMPLE el último día del mes para cada una de las fechas de alta de los empleados del departamento 10:

```
SQL> SELECT FECHA_ALT, LAST_DAY(FECHA_ALT) FROM EMPLE WHERE DEPT_NO=10;

     FECHA_AL            LAST_DAY
     --------            --------
     09/06/91            30/06/91
     17/11/91            30/11/91
     23/01/92            31/01/92
```

(Continúa)



(Continuación)

MONTHS_BETWEEN(fecha1, fecha2). Cálculo de la edad: necesitamos la función "SYSDATE", que devuelve la fecha actual (fecha del sistema) y calculamos los meses transcurridos entre la fecha de hoy y la fecha de nacimiento. Dividimos entre 12 ese resultado y aplicamos la función TRUNC para suprimir decimales:

```
SQL> SELECT TRUNC (MONTHS_BETWEEN (SYSDATE,'18/11/1964') / 12) "Edad actual" FROM DUAL;

    Edad actual
    -----------
             40
```

NEXT_DAY(fecha, cad). Si hoy es jueves 19 de octubre de 2006 (fecha del sistema «sysdate»), ¿qué fecha será el próximo jueves?

```
SQL> SELECT NEXT_DAY(SYSDATE, 'JUEVES')  "Sig. Jueves"  FROM  DUAL;

       Sig. Jue
       --------
       26/10/06
```

4.5 Funciones de conversión

La mayoría de las funciones que hemos visto hasta ahora son funciones de transformación, esto es, cambian los objetos. Hay otras funciones que cambian los objetos de una manera especial, pues transforman un tipo de dato en otro. Las *funciones de conversión* elementales se muestran en la Tabla 4.7. La Tabla 4.8 muestra las máscaras de formato para las fechas y la Tabla 4.9 muestra las máscaras de formatos numéricos.

Función	Propósito
TO_CHAR (fecha, 'formato')	Convierte una _fecha_ (de tipo DATE) a tipo _VARCHAR2_ en el _formato_ especificado. El _formato_ es un cadena de caracteres que puede incluir las máscaras de formato definidas en la *Tabla de control de formato de fechas* (Tabla 4.8), y donde es posible incluir literales definidos por el usuario encerrados entre comillas dobles.
TO_CHAR (numero, 'formato')	Esta función convierte un 'número' (de tipo NUMBER) a tipo VARCHAR2 en el 'formato' especificado. Los formatos numéricos se muestran en la Tabla 4.9.
TO_DATE (cad, 'formato')	Convierte 'cad', de tipo VARCHAR2 o CHAR, a un valor de tipo DATE según el 'formato' especificado.
TO_NUMBER (cadena [,'formato'])	Convierte la 'cadena' a tipo NUMBER según el 'formato' especificado. La cadena ha de contener números, el carácter decimal o el signo menos a la izquierda. No puede haber espacios entre los números, ni otros caracteres.

Tabla 4.7. *Funciones de conversión.*

Máscaras de formato numéricas	
cc o scc	Valor del siglo
y, yyy ó sy,yyy	Año con coma, con o sin signo
yyyy	Año sin signo
yyy	Últimos tres dígitos del año
yy	Últimos dos dígitos del año
y	Último dígito del año
q	Número del trimestre
ww	Número de semana del año
w	Número de semana del mes
mm	Número de mes
ddd	Número de día del año
dd	Número de día del mes
d	Número de día de la semana
hh ó hh12	Hora (1-12)
hh24	Hora (1-24)
mi	Minutos
ss	Segundos
sssss	Segundos transcurridos desde medianoche
j	Juliano

Máscaras de formato de caracteres	
syear ó year	Año en inglés (ej: *nineteen-eighty-two*)
month	Nombre del mes (ENERO)
mon	Abreviatura de tres letras del nombre del mes (ENE)
day	Nombre del día de la semana (LUNES)
dy	Abreviatura de tres letras del nombre del día (LUN)
a.m. o p.m.	Muestra a.m. ó p.m. dependiendo del momento del día
b.c. o a.d.	Indicador para el año (antes de Cristo o después de Cristo)

Tabla 4.8. *Máscaras de control de formatos de fechas.*



Caso práctico

8 **A partir de la tabla EMPLE, obtén la fecha de alta (columna FECHA_ALT) formateada, de manera que aparezca el nombre del mes con todas sus letras (month), el número de día de mes (dd) y el año (yyyy), para aquellos empleados del departamento 10:**

```
SQL> SELECT TO_CHAR (FECHA_ALT, 'month DD, YYYY') "NUEVA FECHA" FROM EMPLE WHERE
DEPT_NO=10;

                    NUEVA      FECHA
                    -------------------
                    junio      09,  1991
                    noviembre  17,  1991
                    enero      23,  1992
```

Hay que hacer una observación: el nombre del mes aparece en minúscula porque en el formato se definió *month* en minúscula, pero también puede aparecer en mayúscula o con la primera letra mayúscula y las siguientes en minúsculas. Las opciones para *month* son las siguientes: si el nombre de mes es Enero, Month produce Enero, month produce enero, MONTH produce ENERO. Lo mismo ocurre con day y dy.

(Continúa)





(Continuación)

Ahora se desea obtener la fecha de alta de forma que aparezca el nombre del mes con tres letras (mon), el número de día del año (ddd), el último dígito del año (y) y los tres últimos dígitos del año (yyy):

```
SQL> SELECT TO_CHAR (FECHA_ALT, 'mon ddd y yyy') "FECHA" FROM EMPLE WHERE DEPT_NO=10;

  FECHA
  -------------
  jun 160 1 991
  nov 321 1 991
  ene 023 2 992
```

Por defecto, el formato para la fecha viene definido por el parámetro **NLS_TERRITORY**, que especifica el idioma para el formato de la fecha, los separadores de miles, el signo decimal y el símbolo de la moneda. Este parámetro se inicializa al arrancar Oracle. Para el idioma español, el valor de este parámetro es: NLS_TERRITORY=SPAIN.

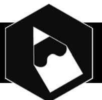
Podemos cambiar el valor por omisión para la fecha con el parámetro **NLS_DATE_FORMAT**, usando la orden **ALTER SESSION**. Por ejemplo, para cambiar el formato de la fecha y que aparezca de la siguiente manera: día/nombre mes/ año hora:minutos:segundos utilizaremos la siguiente sentencia:

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT='DD/month/YYYY
HH24:MI:SS';
Sesión modificada.
SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
---------------------------
03/agosto    /2005 17:20:48
```

También podemos cambiar el lenguaje utilizado para nombrar los meses y los días con el parámetro **NLS_DATE_LANGUAGE**.

Caso práctico

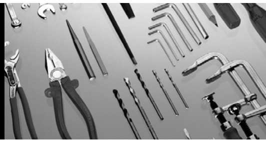


9 **Por defecto, al iniciar nuestra sesión el idioma definido para la fecha es el español. Podemos definir otro idioma, por ejemplo, el francés, de la siguiente manera:** `SQL> ALTER SESSION SET NLS_DATE_LANGUAGE=French;`

Si queremos visualizar el nombre del mes y el día de la semana, éstos aparecerán en francés, con lo que la fecha de hoy nos quedará así:

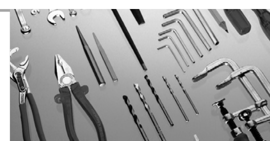
```
SQL> SELECT TO_CHAR(sysdate,'"Hoy es " day "," dd " de " month " de " yyyy')
"Fecha" FROM DUAL;

Fecha
------------------------------------------------
Hoy es  mercredi , 03   de   août       de  2005
```

Elemento	Ejemplo	Descripción
9	999	Devuelve el valor con el número especificado de dígitos. Si es positivo, deja un espacio. Devuelve el valor con el número especificado de dígitos con el signo menos si es negativo. Si el valor tiene ceros a la izquierda, los deja en blanco, excepto si el valor es 0. SQL> SELECT TO_CHAR(1,'999'), TO_CHAR(-1,'999'), TO_CHAR(01,'999') , TO_CHAR(0,'999') FROM DUAL; TO_C TO_C TO_C TO_C 1 -1 1 0
0	9990 999	9990 - Muestra un 0 si el valor es 0. 0999 - Devuelve el valor dejando ceros al principio. SQL> SELECT TO_CHAR(10,'0999'), TO_CHAR(10,'9990') ,TO_CHAR(10,'990090') FROM DUAL; TO_CH TO_CH TO_CHAR 0010 10 0010
$	$9999	Devuelve el valor con el signo dólar a la izquierda. SQL> SELECT TO_CHAR(10,'$9999'),TO_CHAR(10,'$009'), TO_CHAR(10,'99$') FROM DUAL; TO_CHA TO_CH TO_C $10 $010 $10
B	B999	Muestra un espacio en blanco si el valor es 0. Es el formato por omisión. SQL> SELECT TO_CHAR(0,'B999'), TO_CHAR(5,'B999') FROM DUAL; TO_C TO_C 5
MI	999MI	Si el número es negativo, el signo menos sigue al número. Por omisión, el signo se pone a la izquierda. SQL> SELECT TO_CHAR(-55,'999MI'), TO_CHAR (55, '999MI') FROM DUAL; TO_C TO_C 55- 55
S	S999 999S	'S' representa el signo. Devuelve el valor con el signo '+' si el valor es positivo o con el signo '_' si es negativo. SQL> SELECT TO_CHAR(-55,'999S'), TO_CHAR(-55, 'S999'), TO_CHAR(55,'S999'), TO_CHAR(55,'999S') FROM DUAL; TO_C TO_C TO_C TO_C 55- -55 +55 55+
PR	9999PR	Los números negativos se muestran entre estos símbolos: < >. SQL> SELECT TO_CHAR(-55,'9999PR'), TO_CHAR(55, '9999PR') FROM DUAL; TO_CHA TO_CHA <55> 55
D	99D99	Devuelve el carácter decimal en la posición especificada. SQL> SELECT TO_CHAR(34.55,'99D99') FROM DUAL; TO_CHA 34,55

Tabla 4.9. *Tabla de formatos numéricos.*




Elemento	Ejemplo	Descripción

G	9G999	Devuelve el carácter de grupo (carácter de los miles) en la posición especificada.
		`SQL> SELECT TO_CHAR(1234,'9G999') FROM DUAL;`
		`TO_CHA`
		`------`
		` 1.234`
		`SQL> SELECT TO_CHAR(123456.98, '999G999D99') FROM DUAL;`
		`TO_CHAR(123`
		`-----------`
		`123.456,98`

C	C999	Devuelve el símbolo ISO del territorio en la posición especificada.
		`SQL> SELECT TO_CHAR(123,'C999') "ISO" FROM DUAL;`
		`  ISO`
		`------`
		`EUR123`

L	L999
	999L
		Devuelve el símbolo de la moneda local en la posición indicada.
		`SQL> SELECT TO_CHAR(123,'L999') "MONEDA" FROM DUAL;`
		`MONEDA`
		`-------`
		`€ 123`

, (coma)	9,999	Devuelve la coma en la posición especificada (carácter de los miles).
		`SQL> SELECT TO_CHAR(1234,'9,999') FROM DUAL;`
		`TO_CHA`
		`---`
		`1,234`

. (punto)	99.99	Devuelve el punto decimal en la posición especificada.
		`SQL> SELECT TO_CHAR(12.34,'99.99') FROM DUAL;`
		`TO_CHA`
		`------`
		`12.34`
		`SQL> SELECT TO_CHAR(12345.67,'99,999.99') FROM DUAL;`
		`TO_CHAR(12`
		`----------`
		`12,345.67`

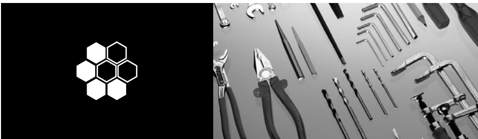
V	999V99	Devuelve el valor multiplicado por 10n, donde 'n' es el número de nueves después 'V'.
		`SQL> SELECT TO_CHAR(123.45,'999V99'), TO_CHAR (123,'999V99') FROM DUAL;`
		`TO_CHA  TO_CHA`
		`------  ------`
		` 12345   12300`

EEEE	9.9EEEE	Devuelve el valor usando notación científica.
		`SQL> SELECT TO_CHAR(12345,'9.9EEEE') FROM DUAL;`
		`TO_CHAR(1`
		`-----`
		`1.2E+04`

RN
 	
rn
	RN	Devuelve el valor en números romanos. 'RN' devuelve el valor en mayúsculas y 'rn', en minúsculas.
		`SQL> SELECT TO_CHAR(12,'RN') , TO_CHAR(12,'rn') FROM DUAL;`
		`TO_CHAR(12,'RN'  TO_CHAR(12,'RN'`
		`---------------  ---------------`
		`      XII              xii`

FM	FM90.9	Devuelve el valor alineado a la izquierda.
		`SQL>SELECT  TO_CHAR(12.8,'FM90.9'),TO_CHAR(12, 'FM99'), TO_CHAR(-12,'FM99') FROM DUAL;`
		`TO_CH  TO_  TO_`
		`-----  ---  ---`
		` 12.8   12  -12`

Tabla 4.9 *(Continuación)*. *Tabla de formatos numéricos.*



Los caracteres devueltos en algunos de estos formatos se especifican inicializando una serie de parámetros. Estos parámetros se muestran en la Tabla 4.10.

Elemento	Elemento	Descripción
NLS_NUMERIC_ CHARACTERS	D, G	Define los caracteres decimal ('D') y separador de los miles ('G'). Formato: NLS_NUMERIC_CHARACTERS= 'DG' ____D: carácter decimal. ____G: separador de miles. Ejemplo:____NLS_NUMERIC_CHARACTERS= ',.' Carácter decimal, la coma, y separador de miles, el punto.
NLS_ISO_CURRENCY	C	Especifica el símbolo del territorio. Para Espaıa el símbolo es 'ESP'.
NLS_CURRENCY	L	Especifica el símbolo de la moneda local. Para España es '€'.

Tabla 4.10. *Parámetros NLS.*

Para cambiar el valor de estos parámetros se utiliza la orden ALTER SESSION. Supongamos, por ejemplo, que queremos definir como carácter de los miles el asterisco (*) y como carácter decimal la barra (/). Hemos de usar el parámetro NLS_NUMERIC_CHARACTERS con ALTER SESSION:

```
SQL> ALTER SESSION SET NLS_NUMERIC_CHARACTERS='/*';
Sesión modificada.
SQL>  SELECT TO_CHAR(12345.67,'999G999D999') FROM DUAL;

TO_CHAR(1234
------------
  12*345/670
```

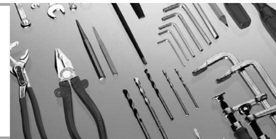
Los valores para el carácter decimal y de los miles permanecerán hasta que el usuario finalice la sesión o hasta que el usuario aplique de nuevo la orden ALTER SESSION para cambiar estos caracteres:

```
SQL> ALTER SESSION SET NLS_NUMERIC_CHARACTERS=',.';
```

Si queremos cambiar el símbolo de la moneda, de manera que, en lugar de '€' aparezca 'Pesetas', usamos el parámetro NLS_CURRENCY. Ejemplo:

```
SQL> ALTER SESSION SET NLS_CURRENCY='PESETAS';
Sesión modificada.
SQL> SELECT TO_CHAR(123,'L999') FROM DUAL;

TO_CHAR(123,'L
--------------
    PESETAS123
```




Caso práctico



10 **TO_NUMBER(cadena [,'formato']).** Suponemos que el carácter decimal es la coma y el carácter separador de los miles, el punto.

```
SQL> SELECT TO_NUMBER('-123456') "NUMERO1", TO_NUMBER('123,99','999D99') "NUMERO2"
FROM DUAL;

          NUMERO1         NUMERO2
        ---------       ---------
          -123456          123,99

SQL> SELECT TO_NUMBER('123.456','999G999') "NO CONVIERTE" FROM DUAL;

     NO CONVIERTE
     ------------
           123456
```

Este ejemplo no convierte porque la cadena '123.456' contiene el carácter que define el separador de los miles (en este ejemplo, el punto), y una cadena válida ha de contener el carácter decimal (en este caso la coma).

TO_DATE(cad , 'formato'). Cambia el formato de la fecha para que aparezca el año con cuatro dígitos:

```
SQL> ALTER SESSION SET NLS_DATE_FORMAT='DD/MM/YYYY';
```

Convertir una cadena a tipo DATE: `SQL> SELECT TO_DATE('01012006') FROM DUAL;`

```
TO_DATE('0
----------
01/01/2006
```

Cuando en la orden TO_DATE no se indica el formato, una cadena de caracteres será convertida a fecha sólo si está en el formato que tenga la fecha del sistema. En el siguiente ejemplo no se convierte la cadena a tipo fecha porque no está en el formato 'DDMMYYYY' definido en la sesión para la fecha: `SQL> SELECT TO_DATE('010106') FROM DUAL;`

```
SELECT TO_DATE('010106') FROM DUAL
       *
ERROR en línea 1:
ORA-01861: el literal no coincide con la cadena de formato
```

Lo correcto sería: `SQL> SELECT TO_DATE('01012006') FROM DUAL;`

Obtén el nombre del mes a partir de la cadena '01012007' (antes hay que convertir la cadena a tipo fecha): `SQL> SELECT TO_CHAR(TO_DATE ('01012007', 'ddmmyyyy'), 'Month') "MES" FROM DUAL;`

```
MES
----------
Enero
```

TO_DATE y TO_CHAR son similares; la diferencia que los separa estriba en que TO_DATE convierte una cadena de caracteres en una fecha y TO_CHAR convierte una fecha en una cadena de caracteres. Ambas pueden utilizar las máscaras de formato de fechas.

4.6 Otras funciones

A continuación, se exponen otras funciones que, por sus características, no se han incluido en los grupos anteriores, aunque no por ello dejan de ser útiles.

DECODE (var, val1, cod1, val2, cod2..., valor_por_defecto)

Esta función sustituye un valor por otro. Si «var» es igual a cualquier valor de la lista («val1», «val2»...), devuelve el correspondiente código («cod1», «cod2»...). En caso contrario se obtiene el valor señalado como valor por defecto («valor_por_defecto»); «val» debe ser un dato del mismo tipo de «var». Ésta es una función IF - THEN - ELSE.

VSIZE (expresión)

Devuelve el número de bytes que ocupa expresión. Si expresión es nulo, la función devuelve nulo.

DUMP (cadena[, formato [, comienzo[, longitud]]])

Esta función visualiza el valor de «cadena», que puede ser un literal o una expresión, en formato de datos interno, en ASCII, octal, decimal, hexadecimal o en formato de carácter. Por defecto, el formato es ASCII o EBCDIC, lo que depende de la máquina. El argumento «formato» puede tener los siguientes valores:

8 devuelve el resultado en octal.
10 devuelve el resultado en decimal.
16 devuelve el resultado en hexadecimal.
17 devuelve el resultado en formato carácter (ASCII o EBCDIC).

«Comienzo» es la posición de inicio de la cadena y «longitud» es el número de caracteres que se van a visualizar.

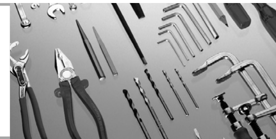
USER

Esta función devuelve el nombre del usuario actual.

UID

Devuelve el identificador del usuario actual. Al crear un usuario, Oracle le asigna un número. Este número identifica a cada usuario y es único en la base de datos.

Aunque USER, UID y SYSDATE se han incluido entre las funciones, realmente son *pseudocolumnas*. Una **pseudocolumna** es una 'columna' que devuelve un valor al seleccionarla, pero que no es una columna actual de una tabla.





Caso práctico



11 **Sea la tabla EMPLE. Seleccionar todas las filas y codificar el OFICIO. Si el oficio es PRESIDENTE, codificar con un 1; si es EMPLEADO, con un 2; en cualquier otro caso, codificar con un 5:**

```
SQL> SELECT APELLIDO, OFICIO, DECODE(UPPER(OFICIO), 'PRESIDENTE', 1, 'EMPLEADO',
2, 5) "Codigo" FROM EMPLE;

     APELLIDO       OFICIO              Codigo
     ----------     ----------      ----------
     SANCHEZ        EMPLEADO                 2
     ARROYO         VENDEDOR                 5
     SALA           VENDEDOR                 5
     JIMENEZ        DIRECTOR                 5
     MARTIN         VENDEDOR                 5
     NEGRO          DIRECTOR                 5
     CEREZO         DIRECTOR                 5
     GIL            ANALISTA                 5
     REY            PRESIDENTE               1
     TOVAR          VENDEDOR                 5
     ALONSO         EMPLEADO                 2
     JIMENO         EMPLEADO                 2
     FERNANDEZ      ANALISTA                 5
     MUÑOZ          EMPLEADO                 2

     14 filas seleccionadas.
```

Calculamos el número de bytes que tiene la columna APELLIDO de la tabla EMPLE para aquellos empleados del departamento 10:

```
SQL> SELECT APELLIDO, VSIZE(APELLIDO) BYTES FROM EMPLE WHERE DEPT_NO=10;

     APELLIDO         BYTES
     ---------      -------
     CEREZO               6
     REY                  3
     MUÑOZ                5
```

Representamos en hexadecimal los caracteres 1 al 4 del APELLIDO 'SALA' de la tabla EMPLE:

```
SQL> SELECT APELLIDO,
DUMP(APELLIDO,16,1,4) FROM EMPLE WHERE APELLIDO LIKE 'SALA';

     APELLIDO       DUMP(APELLIDO,16,1,4)
     ----------     -------------------------
     SALA           Typ=1 Len=4: 53,41,4c,41
```

Visualiza el usuario que está conectado y su identificador:

```
SQL> SELECT USER, UID FROM DUAL;

     USER                UID
     -------          ------
     SCOTT                57
```

La orden SHOW USER muestra el nombre de usuario que está conectado: `SQL> SHOW USER`

Conceptos básicos



Las funciones vistas en la unidad se resumen en el siguiente esquema:

FUNCIONES	**Aritméticas**	De valores simples	ABS(n), CEIL(n), FLOOR(n), MOD(m, n), NVL(valor, expresión), POWER(m, exponente), ROUND(número [,m]), SIGN(valor), SQRT(n), TRUNC(número, [m])
		De grupos de valores	AVG(n), COUNT (* | expresión), MAX(expresión), MIN(expresión), SUM(expresión), VARIANCE (expresión)
		De listas	GREATEST(valor1, valor2..) LEAST (valor1, valor2...)
	De cadenas	Devuelven valores carácter	CHR(n), CONCAT(cad1, cad2) LOWER(cad),UPPER (cad) INITCAP(cad), LPAD(cad1, n [, cad2]), RPAD(cad1, n [, cad2]), LTRIM(cad [, set]), RTRIM(cad [, set]), REPLACE(cad, cadena_búsqueda [,cadena_sustitución]), SUBSTR(cad, m [,n]), TRANSLATE(cad1,cad2, cad3)
		Devuelven valores numéricos	ASCII(cad), INSTR(cad1,cad2 [,comienzo [,m]]), LENGTH(cad)
	Manejo de fechas	SYSDATE , ADD_MONTHS(fecha, n) LAST_DAY(fecha) MONTHS_BETWEEN(fecha1, fecha2) NEXT_DAY(fecha, cad)	
	De conversión	TO_CHAR (fecha, 'formato') TO_CHAR (numero, 'formato') TO_DATE (cad, 'formato') TO_NUMBER(cadena [,'formato'])	
	Otras	DECODE(var,val1,cod1,val2,cod2,.., valor_por_defecto) VSIZE (expresión), DUMP(cadena[, formato [, comienzo[, longitud]]]), USER, UID	





Actividades complementarias



1. Dada la tabla EMPLE, obtén el sueldo medio, el número de comisiones no nulas, el máximo sueldo y el mínimo sueldo de los empleados del departamento 30. Emplea el formato adecuado para la salida para las cantidades numéricas.

2. Visualiza los temas con mayor número de ejemplares de la tabla LIBRERIA y que tengan, al menos, una 'E' (pueden ser un tema o varios).

3. Dada la tabla MISTEXTOS, ¿qué sentencia SELECT se debe ejecutar para tener este resultado?

```
RESULTADO
-------------------------------------
METODOLOGÍA DE LA PROGRAMACIÓN-^-^-^-^-
INFORMÁTICA BÁSICA-^-^-^-^-^-^-^-^-^-
SISTEMAS OPERATIVOS-^-^-^-^-^-^-^-^-^-
SISTEMAS DIGITALES-^-^-^-^-^-^-^-^-^-
MANUAL DE C-^-^-^-^-^-^-^-^-^-^-^-^-^-
```

4. Visualiza los títulos de la tabla MISTEXTOS sin los caracteres punto y comillas, y en minúscula, de dos formas conocidas.

5. Dada la tabla LIBROS, escribe la sentencia SELECT que visualice dos columnas, una con el AUTOR y otra con el apellido del autor.

6. Escribe la sentencia SELECT que visualice las columnas de AUTOR y otra columna con el nombre del autor (sin el apellido) de la tabla LIBROS.

7. A partir de la tabla LIBROS, realiza una sentencia SELECT que visualice en una columna, primero el nombre del autor y, luego, su apellido.

8. A partir de la tabla LIBROS, realiza una sentencia SELECT para que aparezcan los títulos ordenados por su número de caracteres.

9. Dada la tabla NACIMIENTOS, realiza una sentencia SELECT que obtenga la siguiente salida: NOMBRE, FECHANAC, FECHA_FORMATEADA, donde FECHA_FORMATEADA tiene el siguiente formato:

```
"Nació el 12 de mayo de 1982".
```

10. Dada la tabla LIBRERIA, haz una sentencia SELECT que visualice el tema, el último carácter del tema que no sea blanco y el número de caracteres de tema (sin contar los blancos de la derecha) ordenados por tema.

11. A partir de la tabla NACIMIENTOS, visualiza en una columna el NOMBRE seguido de su fecha de nacimiento formateada (quita blancos del nombre).

12. Convierte la cadena '010712' a fecha y visualiza su nombre de mes en mayúsculas.

13. Visualiza aquellos temas de la tabla LIBRERIA cuyos ejemplares sean 7 con el nombre de tema de 'SEVEN'; el resto de temas que no tengan 7 ejemplares se visualizarán como están.

14. A partir de la tabla EMPLE, obtén el apellido de los empleados que lleven más de 15 años trabajando.

15. Selecciona el apellido de los empleados de la tabla EMPLE que lleven más de 16 años trabajando en el departamento 'VENTAS'.

16. Visualiza el apellido, el salario y el número de departamento de aquellos empleados de la tabla EMPLE cuyo salario sea el mayor de su departamento.

17. Visualiza el apellido, el salario y el número de departamento de aquellos empleados de la tabla EMPLE cuyo salario supere a la media en su departamento.

Cláusulas avanzadas de selección

5

En esta unidad aprenderás a:

1. **Elegir las cláusulas necesarias para realizar la agrupación de filas.**
2. **Distinguir cuándo usar las cláusulas WHERE y HAVING.**
3. **Emplear correctamente los OUTER-JOIN o combinación externa.**
4. **Utilizar de manera correcta los operadores de conjuntos en una sentencia SELECT.**





5.1 Introducción

En esta unidad vamos a continuar haciendo consultas a la base de datos. Nos ocuparemos de nuevas cláusulas que acompañan a la sentencia SELECT y que permiten llegar a consultas más complejas. Veremos las órdenes que nos permiten agrupar filas de una tabla según alguna condición, o sin ninguna condición, para obtener algún resultado referente a ese grupo de filas agrupadas. Un ejemplo de esta agrupación es averiguar cuál es la suma de salarios por cada departamento de la tabla EMPLE.

También nos ocuparemos de otro tipo de combinaciones de tablas: la que nos permite seleccionar algunas filas de una tabla aunque éstas no tengan su correspondencia con la otra tabla. Estudiaremos cómo podemos combinar los resultados de varias sentencias SELECT utilizando operadores de conjuntos.

5.2 Agrupación de elementos. GROUP BY y HAVING

Hasta ahora hemos utilizado la **orden SELECT** para recuperar filas de una tabla y la **cláusula WHERE** para seleccionar el número de filas que se recuperan. También hemos empleado funciones de grupo para trabajar con conjuntos de filas. Se puede pensar en estos conjuntos como si fueran un grupo; así calculamos, por ejemplo, el salario medio de todos los empleados: `SELECT AVG(SALARIO) FROM EMPLE;` o la suma de todos los salarios: `SELECT SUM(SALARIO) FROM EMPLE;`.

Pero, a veces, nos interesa consultar los datos según grupos determinados. Así, para saber cuál es el salario medio de cada departamento de la tabla EMPLE, las cláusulas que conocemos hasta ahora no son suficientes. Necesitamos realizar un agrupamiento por departamento. Para ello utilizaremos la cláusula **GROUP BY**. La consulta sería la siguiente:

```
SELECT DEPT_NO, AVG(SALARIO) FROM EMPLE GROUP BY DEPT_NO;
```

La sentencia SELECT posibilita agrupar uno o más conjuntos de filas. El agrupamiento se lleva a cabo mediante la cláusula GROUP BY por las columnas especificadas y en el orden especificado. Éste es su formato:

```
SELECT ...
FROM ...
GROUP BY columna1, columna2, columna3,...
HAVING condición
ORDER BY ...
```

Los datos seleccionados en la sentencia SELECT que lleva el GROUP BY deben ser: una constante, una función de grupo (SUM, COUNT, AVG, ...), una columna expresada en el GROUP BY.

La cláusula **GROUP BY** sirve para calcular propiedades de uno o más conjuntos de filas. Además, si se selecciona más de un conjunto de filas, GROUP BY controla que las filas de la tabla original sean agrupadas en una temporal. Del mismo modo que existe la condición de búsqueda **WHERE** para filas individuales, también hay una condición de búsqueda para grupos de filas: **HAVING.** La cláusula HAVING se emplea para controlar cuál de los conjuntos de filas se visualiza. Se evalúa sobre la tabla que devuelve el GROUP BY. No puede existir sin GROUP BY.



Caso práctico

1 Visualiza a partir de la tabla EMPLE el número de empleados que hay en cada departamento.

Para hacer esta consulta, tenemos que agrupar las filas de la tabla EMPLE por departamento (GROUP BY DEPT_NO) y contarlas (COUNT(*)). La consulta es la siguiente:

```
SQL> SELECT DEPT_NO, COUNT(*) FROM EMPLE GROUP BY DEPT_NO;

   DEPT_NO    COUNT(*)
   -------  ----------
        10           3
        20           5
        30           6
```

COUNT es una función de grupo y da información sobre un grupo de filas, no sobre filas individuales de la tabla. La cláusula GROUP BY DEPT_NO obliga a COUNT a contar las filas que se han agrupado por cada departamento.

Si en la consulta anterior sólo queremos visualizar los departamentos con más de 4 empleados, tendríamos que escribir lo siguiente:

```
SQL> SELECT DEPT_NO, COUNT(*) FROM EMPLE GROUP BY DEPT_NO HAVING COUNT(*) > 4;

   DEPT_NO    COUNT(*)
   -------  ----------
        20           5
        30           6
```



Actividades propuestas

1 Visualiza los departamentos en los que el salario medio es mayor o igual que la media de todos los salarios.

La cláusula **HAVING** es similar a la cláusula WHERE, pero trabaja con grupos de filas; pregunta por una característica de grupo, es decir, pregunta por los resultados de las funciones de grupo, lo cual WHERE no puede hacer. En el ejemplo anterior se visualizan las filas cuyo número de empleados sea mayor de 4 (**HAVING** `COUNT(*) > 4`).





Si queremos ordenar la salida descendentemente por número de empleados, obteniendo el resultado de la Tabla 5.1, utilizamos la siguiente orden ORDER BY:

```
SQL> SELECT DEPT_NO, COUNT(*) FROM EMPLE
  2  GROUP BY DEPT_NO
  3  HAVING COUNT(*) > 4
  4  ORDER BY COUNT(*) DESC;
```

DEPT_NO	COUNT(*)
30	6
20	5

Tabla 5.1. *HAVING y ORDER BY.*

Cuando usamos la cláusula ORDER BY con columnas y funciones de grupo hemos de tener en cuenta que ésta se ejecuta detrás de las cláusulas WHERE, GROUP BY y HAVING. En ORDER BY podemos especificar funciones de grupo, columnas de GROUP BY o su combinación.

La evaluación de las cláusulas en tiempo de ejecución se efectúa en el siguiente orden:

```
WHERE    Selecciona las filas.
GROUP BY Agrupa estas filas.
HAVING   Filtra los grupos. Selecciona y elimina los grupos.
ORDER BY Clasifica la salida. Ordena los grupos.
```

Caso práctico

2 **Consideramos las tablas EMPLE y DEPART. Obtén la suma de salarios, el salario máximo y el salario mínimo por cada departamento; la salida de los cálculos debe estar formateada:**

```
SQL> SELECT DEPT_NO, TO_CHAR (SUM(SALARIO), '99G999D99') "Suma",
  2  TO_CHAR (MAX(SALARIO), '99G999D99') "Máximo",
  3  TO_CHAR (MIN(SALARIO), '99G999D99') "Mínimo"
  4  FROM EMPLE GROUP BY DEPT_NO;

   DEPT_NO         Suma         Máximo       Mínimo
----------   ----------     ----------   ----------
        10     8.675,00       4.100,00     1.690,00
        20    11.370,00       3.000,00     1.040,00
        30    10.415,00       3.005,00     1.335,00
```

Calcula el número de empleados que realizan cada OFICIO en cada DEPARTAMENTO. Los datos que se visualizan son: departamento, oficio y número de empleados. Necesitamos agrupar por departamento y dentro de cada departamento, por oficio:

```
SQL> SELECT DEPT_NO, OFICIO, COUNT(*) FROM EMPLE GROUP BY DEPT_NO, OFICIO;

   DEPT_NO    OFICIO        COUNT(*)
----------    -----------  ----------
        10    DIRECTOR              1
        10    EMPLEADO              1
        10    PRESIDENTE            1
        20    ANALISTA              2
```

(Continúa)

(Continuación)

```
            20       DIRECTOR               1
            20       EMPLEADO               2
            30       DIRECTOR               1
            30       EMPLEADO               1
            30       VENDEDOR               4

9 filas seleccionadas.
```

Busca el número máximo de empleados que hay en algún departamento:

```
SQL> SELECT  MAX(COUNT(*)) "Máximo"  FROM EMPLE GROUP BY DEPT_NO;

    Máximo
----------
         6
```



Actividades propuestas

2 **Obtén los nombres de departamentos que tengan más de 4 personas trabajando.**

Visualiza el número de departamento, el nombre de departamento y el número de empleados del departamento con más empleados.

5.3 Combinación externa (OUTER JOIN)

Ya hemos tratado el concepto de *combinación de tablas*. Existe una variedad de combinación de tablas que se llama **OUTER JOIN** y que nos permite seleccionar algunas filas de una tabla aunque éstas no tengan correspondencia con las filas de la otra tabla con la que se combina. El formato es el siguiente:

```
SELECT tabla1.colum1, tabla1.colum2, tabla2.colum1
FROM tabla1, tabla2
WHERE tabla1.colum1 = tabla2.colum1(+);
```

Esta SELECT seleccionará todas las filas de la tabla *tabla1*, aunque no tengan correspondencia con las filas de la tabla *tabla2*; se denota con el símbolo **(+)** detrás de la columna de la *tabla2* (que es la tabla donde no se encuentran las filas) en la cláusula WHERE. Se obtendrá una fila con las columnas de *tabla1* y el resto de columnas de la *tabla2* se rellena con NULL.




Veamos el funcionamiento del OUTER JOIN con un ejemplo.

Sean las tablas EMPLE Y DEPART, comprobamos su contenido ejecutando desde SQL:

`SELECT * FROM DEPART; y SELECT * FROM EMPLE ORDER BY DEPT_NO;`

Si nos fijamos en la tabla EMPLE, vemos que el departamento 40 no existe, sin embargo, en DEPART sí existe.

Queremos realizar una consulta donde se visualice el número de departamento, el nombre y el número de empleados que tiene. Para hacer esta consulta combinamos las dos tablas:

```
SQL> SELECT D.DEPT_NO, DNOMBRE, COUNT(E.EMP_NO)
  2  FROM EMPLE E, DEPART D
  3  WHERE E.DEPT_NO = D.DEPT_NO
  4  GROUP BY D.DEPT_NO, DNOMBRE;

     DEPT_NO      DNOMBRE          COUNT(E.EMP_NO)
     -------      --------------   ---------------
          10      CONTABILIDAD                   3
          20      INVESTIGACION                  5
          30      VENTAS                         6
```

Observamos que con esta sentencia sólo aparecen los departamentos que tienen empleados. El departamento 40 se pierde. Para que aparezca este departamento, que no se corresponde con ninguna fila en la tabla EMPLE, usamos la combinación externa, para lo cual especificamos un **(+)** a continuación del nombre de la columna de la tabla en la que no aparece el departamento (la tabla EMPLE):

```
SQL>  SELECT D.DEPT_NO, DNOMBRE, COUNT(E.EMP_NO)
  2  FROM EMPLE E, DEPART D
  3  WHERE E.DEPT_NO (+) = D.DEPT_NO
  4  GROUP BY D.DEPT_NO, DNOMBRE;

      EPT_NO      DNOMBRE          COUNT(E.EMP_NO)
     -------      --------------   ---------------
          10      CONTABILIDAD                   3
          20      INVESTIGACION                  5
          30      VENTAS                         6
          40      PRODUCCION                     0
```

Actividades propuestas



3 **Analiza lo que ocurre si en lugar de COUNT(E.EMP_NO) ponemos COUNT(*) en la sentencia SELECT anterior. Analiza también lo que ocurre si a la derecha de SELECT ponemos E.DEPT_NO en lugar de D.DEPT_NO.**

5.4 Operadores UNION, INTERSECT y MINUS

Los **operadores relacionales UNION, INTERSECT** y **MINUS** son *operadores de conjuntos*. Los **conjuntos** son las filas resultantes de cualquier sentencia SELECT válida que permiten combinar los resultados de varias SELECT para obtener un único resultado.

Supongamos que tenemos dos listas de centros de enseñanza de una ciudad y que queremos enviar a esos centros una serie de paquetes de libros.

Dependiendo de ciertas características de los centros, podemos enviar libros a todos los centros de ambas listas (UNION), a los centros que estén en las dos listas (INTERSECT) o a los que están en una lista y no están en la otra (MINUS). El formato de SELECT con estos operadores es el siguiente:

```
SELECT ... FROM ... WHERE...
Operador_de_conjunto
SELECT ... FROM ... WHERE...
```

A. Operador UNION

El **operador UNION** combina los resultados de dos consultas. Las filas duplicadas que aparecen se reducen a una fila única. Éste es su formato:

```
SELECT COL1, COL2, ... FROM TABLA1 WHERE CONDICION
UNION
SELECT COL1, COL2, ... FROM TABLA2 WHERE CONDICION;
```



Caso práctico

3 Disponemos de tres tablas:

ALUM contiene los nombres de alumnos que se han matriculado este curso en el centro, NUEVOS contiene los nombres de los alumnos que han reservado plaza para el próximo curso y ANTIGUOS contiene los nombres de antiguos alumnos del centro. La descripción es la misma para las tres:

```
SQL> DESC ALUM

     Nombre          ¿Nulo?        Tipo
     ----------      -------       ---------
     NOMBRE                        VARCHAR2(20)
     EDAD                          NUMBER(2)
     LOCALIDAD                     VARCHAR2(15)
```

(Continúa)




(Continuación)

Visualizamos el contenido de las tablas:

```
SQL> SELECT * FROM ALUM;
SQL> SELECT * FROM NUEVOS;
SQL> SELECT * FROM ANTIGUOS;
```

Visualiza los nombres de los alumnos actuales y de los futuros alumnos. Obtenemos los nombres de alumnos que aparezcan en las tablas ALUM y NUEVOS de la siguiente manera:

```
SQL> SELECT NOMBRE FROM ALUM UNION SELECT NOMBRE FROM NUEVOS;

     NOMBRE
     ----------
     ANA
     ERNESTO
     JUAN
     LUISA
     MAITE
     MARÍA
     PEDRO
     RAQUEL
     SOFÍA

9 filas seleccionadas.
```

UNION ALL combina los resultados de dos consultas. Cualquier duplicación de filas que se dé en el resultado final aparecerá en la consulta. En la consulta anterior, usando UNION ALL aparecerán nombres duplicados, como podemos observar en la Tabla 5.2:

```
SQL> SELECT NOMBRE FROM ALUM UNION ALL SELECT NOMBRE FROM
NUEVOS;
```

NOMBRE
JUAN
PEDRO
ANA
LUISA
MARíA
ERNESTO
RAQUEL
JUAN
MAITE
SOFíA
ANA
ERNESTO
12 filas seleccionadas.

Tabla 5.2. *Consulta con UNION ALL.*

B. Operador INTERSECT

El **operador INTERSECT** devuelve las filas que son iguales en ambas consultas. Todas las filas duplicadas serán eliminadas antes de la generación del resultado final. Su formato es:

```
SELECT COL1, COL2, ... FROM TABLA1 WHERE CONDICION
INTERSECT
SELECT COL1, COL2, ... FROM TABLA2 WHERE CONDICION;
```



Caso práctico

4 **Obtén los nombres de alumnos que están actualmente en el centro y que estuvieron en el centro hace ya un tiempo.**

Necesitamos los nombres que están en la tabla ALUM y que, además, aparezcan en la tabla de ANTIGUOS alumnos:

```
SQL> SELECT NOMBRE FROM ALUM INTERSECT SELECT NOMBRE FROM ANTIGUOS;

  NOMBRE
  ---------
  ERNESTO
  MARÍA
```



Actividades propuestas

4 **Esta consulta también se puede hacer usando el operador IN. Escribe la consulta anterior utilizando el operador IN.**

C. Operador MINUS

El **operador MINUS** devuelve aquellas filas que están en la primera SELECT y no en la segunda. Las filas duplicadas del primer conjunto se reducirán a una fila única antes de que empiece la comparación con el otro conjunto. Su formato es éste:

```
SELECT COL1, COL2, ... FROM TABLA1 WHERE CONDICION
MINUS
SELECT COL1, COL2, ... FROM TABLA2 WHERE CONDICION;
```



Caso práctico

5 **Obtén los nombres y la localidad de alumnos que están actualmente en el centro y que nunca estuvieron anteriormente en él.**

Ordenamos la salida por LOCALIDAD. Necesitamos los nombres que están en la tabla ALUM y que, además, no aparezcan en la tabla de ANTIGUOS alumnos:

```
SQL> SELECT NOMBRE, LOCALIDAD FROM ALUM MINUS SELECT NOMBRE, LOCALIDAD FROM ANTIGUOS
ORDER BY LOCALIDAD;

       NOMBRE              LOCALIDAD
       ------------        -------------
       ANA                 ALCALÁ
       JUAN                COSLADA
       PEDRO               COSLADA
       RAQUEL              TOLEDO
       LUISA               TORREJÓN
```

(Continúa)




(Continuación)

Esta consulta también se puede hacer usando el **operador NOT IN:**

```
SQL> SELECT NOMBRE, LOCALIDAD FROM ALUM WHERE NOMBRE NOT IN (SELECT NOMBRE FROM ANTIGUOS) ORDER BY LOCALIDAD;
```

– Seleccionamos los nombres de la tabla ALUM que estén en NUEVOS y no en ANTIGUOS: `SQL> SELECT NOMBRE FROM ALUM INTERSECT SELECT NOMBRE FROM NUEVOS MINUS SELECT NOMBRE FROM ANTIGUOS;`

  Esta misma consulta se puede obtener con el **operador IN:** `SQL> SELECT NOMBRE FROM ALUM WHERE NOMBRE IN (SELECT NOMBRE FROM NUEVOS MINUS SELECT NOMBRE FROM ANTIGUOS);`

– Seleccionamos los nombres de la tabla ALUM que estén en NUEVOS o en ANTIGUOS: `SQL> SELECT NOMBRE FROM ALUM WHERE NOMBRE IN (SELECT NOMBRE FROM NUEVOS UNION SELECT NOMBRE FROM ANTIGUOS);`

  La consulta también se puede hacer sin usar UNION, pero recurriendo al **operador OR** (hemos de tener en cuenta que el uso de operadores de conjunto en lugar de IN, AND y OR es decisión del programador): `SQL> SELECT NOMBRE FROM ALUM WHERE NOMBRE IN (SELECT NOMBRE FROM NUEVOS) OR NOMBRE IN (SELECT NOMBRE FROM ANTIGUOS);`

Actividades propuestas



5 **Visualiza los nombres de los alumnos de la tabla ALUM que aparezcan en alguna de estas tablas: NUEVOS y ANTIGUOS.**

Escribe las distintas formas en que se puede poner la consulta anterior llegando al mismo resultado.

D. Reglas para la utilización de operadores de conjuntos

Los operadores de conjuntos pueden encadenarse. Los conjuntos se evaluan de izquierda a derecha; para forzar precedencia se pueden utilizar paréntesis.

Estos operadores se pueden manejar con consultas de diferentes tablas, siempre que se apliquen las siguientes reglas:

- Las columnas de las dos consultas se relacionan en orden, de izquierda a derecha.
- Los nombres de columna de la primera sentencia SELECT no tienen por qué ser los mismos que los nombres de columna de la segunda.
- Las SELECT necesitan tener el mismo número de columnas.
- Los tipos de datos deben coincidir, aunque la longitud no tiene que ser la misma.

Conceptos básicos



Para generar los resultados de una instrucción SELECT hay que seguir estos pasos:

1. **FROM:** Se evalúa la cláusula FROM para comprobar la tabla o tablas a usar.
2. **WHERE:** Si hay cláusula WHERE aplicar condición de búsqueda a cada fila.
3. **GROUP BY:** Agrupa estas filas seleccionadas con WHERE.
4. **HAVING:** Aplica la condición de búsqueda a los grupos de filas. Selecciona los grupos.
5. **SELECT:** Extrae las columnas de cada fila que queda.
6. **SELECT DISTINCT:** Elimina filas duplicadas.
7. **ORDER BY:** Clasifica los resultados.
8. Si la instrucción es una UNION, INTERSECT o MINUS se repiten los pasos para cada SELECT.

Aquí tienes un resumen del formato de la sentencia SELECT con las cláusulas vistas hasta ahora:

```
SELECT [ALL|DISTINCT] {expresión1, expresión2, ..., expresiónn | * }
FROM   {tabla1 [,tabla2, ...., tablan] }
[WHERE condición_de_búsqueda]
[GROUP BY expresión [,expresión ...]]
[HAVING condición_de_búsqueda]
[{UNION | INTERSECT | MINUS} SELECT ...]
[ORDER BY {expresión | posición_de_columna}
[DESC|ASC]
[, {expresión | posición_de_columna}
[DESC|ASC] ]  ...];
```

Donde:

- Una expresión puede ser una constante, una referencia a la columna de una tabla o una expresión aritmética.
- Los corchetes ([]) indican un elemento opcional encerrado entre ellos.
- Las barras verticales (|) indican una elección entre dos o más elementos.
- Las llaves ({ }) indican una elección entre elementos requeridos.





Actividades complementarias



Tablas EMPLE Y DEPART

1. **Partiendo de la tabla EMPLE, visualiza por cada oficio de los empleados del departamento 'VENTAS' la suma de salarios.**

2. **Selecciona aquellos apellidos de la tabla EMPLE cuyo salario sea igual a la media del salario en su departamento.**

3. **A partir de la tabla EMPLE, visualiza el número de empleados de cada departamento cuyo oficio sea 'EMPLEADO'.**

4. **Desde la tabla EMPLE, visualiza el departamento que tenga más empleados cuyo oficio sea 'EMPLEADO'.**

5. **A partir de las tablas EMPLE y DEPART, visualiza el número de departamento y el nombre de departamento que tenga más empleados cuyo oficio sea 'EMPLEADO'.**

6. **Busca los departamentos que tienen más de dos personas trabajando en la misma profesión.**

Tablas ALUM, ANTIGUOS Y NUEVOS

7. **Visualiza los nombres de los alumnos de la tabla ALUM que aparezcan en estas dos tablas: ANTIGUOS y NUEVOS.**

8. **Escribe las distintas formas en que se puede poner la consulta anterior llegando al mismo resultado.**

9. **Visualiza aquellos nombres de la tabla ALUM que no estén en la tabla ANTIGUOS ni en la tabla NUEVOS.**

Tablas PERSONAL, PROFESORES Y CENTROS (hacer DESC de las tablas)

10. **Realiza una consulta en la que aparezca por cada centro y en cada especialidad el número de profesores. Si el centro no tiene profesores, debe aparecer un 0 en la columna de número de profesores. Las columnas a visualizar son: nombre de centro, especialidad y número de profesores.**

11. **Obtén por cada centro el número de empleados. Si el centro carece de empleados, ha de aparecer un 0 como número de empleados.**

12. **Obtén la especialidad con menos profesores.**

Tablas BANCOS, SUCURSALES, CUENTAS y MOVIMIENTOS (hacer DESC de las tablas)

TABLA BANCOS: Contiene los datos de los bancos, una fila por cada banco. Un banco se identifica por el *COD_BANCO*.

TABLA SUCURSALES: Contiene los datos de las sucursales. Una fila por sucursal. Cada sucursal se identifica por el *COD_BANCO+COD_SUCUR*.

TABLA CUENTAS: Contiene los datos de las cuentas abiertas en las sucursales de los bancos. Una cuenta se identifica por las columnas *COD_BANCO+COD_SUCUR+NUM_CTA*. Contiene los saldos de las cuentas. SALDO_DEBE contiene la suma de Reintegros y SALDO_HABER la suma de Ingresos.

TABLA MOVIMIENTOS: Contiene los movimientos de las cuentas. Una fila representa un movimiento de una cuenta. La columna TIPO_MOV puede ser I (ingreso) o R (reintegro).

13. **Obtén el banco con más sucursales. Los datos a obtener son:**

Nombre Banco	NºSucursales
xxxxxx	xx

14. **El saldo actual de los bancos de 'GUADALAJARA', 1 fila por cada banco:**

Nombre Banco	Saldo Debe	Saldo Haber
xxxxxx	xx,xx	xx,xx

15. **Datos de la cuenta o cuentas con más movimientos:**

Nombre Cta	Nºmovimientos
xxxxxx	xx

16. **El nombre de la sucursal que haya tenido más suma de reintegros:**

Nombre sucursal	Suma Reintegros
xxxxxx	xx,xx

Manipulación de datos. INSERT, UPDATE y DELETE

6

En esta unidad aprenderás a:

1. Manejar con fluidez las órdenes para insertar, modificar y eliminar filas de una tabla.
2. Utilizar la orden INSERT.
3. Usar la orden UPDATE.
4. Manejar la orden DELETE.
5. Entender los conceptos de transacción, COMMIT y ROLLBACK.




6.1 Introducción

Hasta ahora nos hemos dedicado a consultar datos de la base de datos mediante la sentencia SELECT. Hemos trabajado y seleccionado datos de tablas. Ha llegado el momento de cambiar los datos de las tablas de la base de datos. En esta unidad aprenderemos a insertar nuevas filas en una tabla, a actualizar los valores de las columnas en las filas y a borrar filas enteras.

6.2 Inserción de datos. Orden INSERT

Empezamos la manipulación de datos de una tabla con la orden **INSERT**. Con ella se añaden filas de datos en una tabla. El formato de esta orden es el siguiente:

```
INSERT INTO NombreTabla [(columna [, columna] ...)]
VALUES (valor [, valor] ...);
```

NombreTabla es la tabla en la que se van a insertar las filas.

[(columna [, columna] ...)] representa la columna o columnas donde se van a introducir valores. Si las columnas no se especifican en la cláusula INSERT, se consideran, por defecto, todas las columnas de la tabla.

(valor [, valor] ...) representa los valores que se van a dar a las columnas. Éstos se deben corresponder con cada una de las columnas que aparecen; además, deben coincidir con el tipo de dato definido para cada columna. Cualquier columna que no se encuentre en la lista de columnas recibirá el valor NULL, siempre y cuando no esté definida como NOT NULL, en cuyo caso INSERT fallará. Si no se da la lista de columnas, se han de introducir valores en todas las columnas.

Es posible introducir los valores directamente en la sentencia u obtenerlos a partir de la información existente en la base de datos mediante la inclusión de una consulta haciendo uso de la sentencia SELECT.

Caso práctico



1 Consideremos la tabla PROFESORES, cuya descripción y contenido son:

```
SQL> DESC PROFESORES

Nombre                                     ¿Nulo?     Tipo
-----------------------------------------  --------   -------------
COD_CENTRO                                 NOT NULL   NUMBER(4)
DNI                                                   NUMBER(10)
APELLIDOS                                             VARCHAR2(30)
ESPECIALIDAD                                          VARCHAR2(16)
```

(Continúa)

(Continuación)

```
SQL> SELECT * FROM PROFESORES;
   COD_CENTRO          DNI    APELLIDOS                         ESPECIALIDAD
   ----------   ----------    ------------------------------    --------------
           10      1112345    Martínez Salas, Fernando          INFORMÁTICA
           10      4123005    Bueno Zarco, Elisa                MATEMÁTICAS
           10      4122025    Montes García, M.Pilar            MATEMÁTICAS
           15      9800990    Ramos Ruiz, Luis                  LENGUA
           15      1112345    Rivera Silvestre, Ana             DIBUJO
           15      8660990    De Lucas Fdez, M.Angel            LENGUA
           22      7650000    Ruiz Lafuente, Manuel             MATEMÁTICAS
           45     43526789    Serrano Laguía, María             INFORMÁTICA

8 filas seleccionadas.
```

Damos de alta a una profesora con estos apellidos y nombre: 'Quiroga Martín, A. Isabel', de la especialidad 'INFORMÁTICA' y con el código de centro 45. Las columnas a las que damos valores son: APELLIDOS, ESPECIALIDAD y COD_CENTRO:

```
SQL> INSERT INTO PROFESORES (APELLIDOS, ESPECIALIDAD, COD_CENTRO)
  2  VALUES ('Quiroga Martín, A.Isabel', 'INFORMÁTICA', 45);
  1  fila creada.
```

Al ejecutar la sentencia, Oracle emite un mensaje `(1 fila creada.)` con el que indica que la fila se ha insertado correctamente. Observamos en esta sentencia que:

- Las columnas a las que damos valores se identifican por su nombre.
- La asociación columna-valor es posicional.
- Los valores que se dan a las columnas deben coincidir con el tipo de dato definido en la columna.
- Los valores constantes de tipo carácter han de ir encerrados entre comillas simples (' ') (los de tipo fecha, también).

Ahora, el contenido de la tabla PROFESORES tendrá una fila más:

```
SQL> SELECT * FROM PROFESORES;
   COD_CENTRO          DNI    APELLIDOS                         ESPECIALIDAD
   ----------   ----------    ------------------------------    --------------
           10      1112345    Martínez Salas, Fernando          INFORMÁTICA
           10      4123005    Bueno Zarco, Elisa                MATEMÁTICAS
           10      4122025    Montes García, M.Pilar            MATEMÁTICAS
           15      9800990    Ramos Ruiz, Luis                  LENGUA
           15      1112345    Rivera Silvestre, Ana             DIBUJO
           15      8660990    De Lucas Fdez, M.Angel            LENGUA
           22      7650000    Ruiz Lafuente, Manuel             MATEMÁTICAS
           45     43526789    Serrano Laguía, María             INFORMÁTICA
           45                 Quiroga Martín, A.Isabel          INFORMÁTICA

 9 filas seleccionadas.
```

(Continúa)





(Continuación)

Las columnas para las que no dimos valores aparecen como nulos; en este caso, la columna DNI.

Insertamos a un profesor que no tiene código de centro asignado, de apellidos y nombre 'Seco Jiménez, Ernesto' y de la especialidad 'LENGUA'. Las columnas a las que damos valores son APELLIDOS y ESPECIALIDAD:

```
SQL> INSERT INTO PROFESORES (APELLIDOS, ESPECIALIDAD)
  2  VALUES ('Seco Jiménez, Ernesto', 'LENGUA');
INSERT INTO PROFESORES (APELLIDOS, ESPECIALIDAD)
*
ERROR en línea 1:
ORA-01400: no se puede realizar una inserción NULL en
("SCOTT"."PROFESORES"."COD_CENTRO")
```

Observamos que aparece un mensaje de error que indica que no se puede insertar una columna con valor NULO en la tabla si en su definición se ha especificado NOT NULL (COD_CENTRO está definido como NOT NULL).

Insertamos a un profesor de apellidos y nombre 'Gonzalez Sevilla, Miguel A.' en el código de centro 22, con DNI 23444800 y de la especialidad de 'HISTORIA':

```
SQL> INSERT INTO PROFESORES
  2  VALUES (22, 23444800, 'González Sevilla, Miguel A.', 'HISTORIA');
  1  fila creada.
```

No es necesario especificar el nombre de las columnas ya que las hemos dado un valor en la fila que insertamos. Este valor ha de ir en el mismo orden en que las columnas estén definidas en la tabla.

Actividades propuestas



1 **Escribe la sentencia INSERT anterior de otra manera.**

Inserta un profesor cuya especialidad supere los 16 caracteres de longitud. Comenta el resultado.

A. Inserción con SELECT

Hasta el momento sólo hemos insertado una fila, pero si añadimos a INSERT una consulta, es decir, una sentencia SELECT, se añaden tantas filas como devuelva la consulta. El formato de INSERT con SELECT es el siguiente:

```
INSERT    INTO NombreTabla1 [(columna [, columna] ...)]
     SELECT {columna [, columna] ... | *}
     FROM NombreTabla2 [CLÁUSULAS DE SELECT];
```

Si las columnas no se especifican en la cláusula INSERT, por defecto, se consideran todas las columnas de la tabla.



Caso práctico

2 **Disponemos de la tabla EMPLE30, cuya descripción es la misma que la de la tabla EMPLE. Insertamos los datos de los empleados del departamento 30:**

```
SQL> INSERT INTO EMPLE30
  2 (EMP_NO, APELLIDO, OFICIO, DIR, FECHA_ALT, SALARIO, COMISION, DEPT_NO)
  3  SELECT
  4  EMP_NO, APELLIDO, OFICIO, DIR, FECHA_ALT, SALARIO, COMISION, DEPT_NO
  5  FROM EMPLE
  6  WHERE DEPT_NO=30;

6 filas creadas.
```

Como las tablas EMPLE y EMPLE30 tienen la misma descripción, no es preciso especificar las columnas, siempre y cuando queramos dar valores a todas las columnas. Esta sentencia daría el mismo resultado:

```
SQL> INSERT INTO EMPLE30 SELECT * FROM EMPLE WHERE DEPT_NO=30;
  6  filas creadas.
```

Disponemos de la tabla NOMBRES, que tiene la siguiente descripción:

```
SQL> DESC NOMBRES
    Nombre                                         ¿Nulo?       Tipo
    -----------------------------------------      --------     ---------------
    NOMBRE                                                      VARCHAR2(15)
    EDAD                                                        NUMBER(2)
```

Insertamos en la tabla NOMBRES, en la columna NOMBRE, el APELLIDO de los empleados de la tabla EMPLE que sean del departamento 20: `INSERT INTO NOMBRES (NOMBRE) SELECT APELLIDO FROM EMPLE WHERE DEPT_NO=20;`

Si, al insertar los apellidos, alguno supera la longitud para la columna NOMBRE de la tabla NOMBRES, no se insertará y aparecerá un error.

Insertar un empleado de apellido 'GARCÍA' , con número de empleado 1111, en la tabla EMPLE, en el departamento con mayor número de empleados. La fecha de alta será la actual; inventamos el resto de los valores. En primer lugar, vamos a averiguar qué sentencia SELECT calcula el departamento con más empleados: `SELECT DEPT_NO FROM EMPLE GROUP BY DEPT_NO HAVING COUNT(*) = (SELECT MAX(COUNT(*)) FROM EMPLE GROUP BY DEPT_NO);`

```
      DEPT_NO
    ---------
           30
```

Ahora hacemos la inserción en la tabla EMPLE, teniendo en cuenta la SELECT anterior:

```
INSERT INTO EMPLE SELECT DISTINCT 1111, 'GARCIA', 'ANALISTA', 7566,
                       SYSDATE, 2000, 120, DEPT_NO
 FROM EMPLE WHERE DEPT_NO= (SELECT DEPT_NO FROM EMPLE GROUP BY DEPT_NO
   HAVING COUNT(*)= (SELECT MAX(COUNT(*)) FROM EMPLE GROUP BY DEPT_NO));
```

(Continúa)




(Continuación)

Al hacer la inserción sólo desconocemos el valor de la columna DEPT_NO, que es el que devuelve la SELECT; el resto de valores, como APELLIDO, OFICIO y EMP_NO, los conocemos y, por tanto, los ponemos directamente en la sentencia SELECT. La cláusula DISTINCT es necesaria, ya que sin ella se insertarían tantas filas como empleados haya en el departamento con mayor número de empleados. Si ejecutamos la sentencia sin DISTINCT insertará más de una fila.

Insertar un empleado de apellido 'QUIROGA', con número de empleado 1112, en la tabla EMPLE. Los restantes datos del nuevo empleado serán los mismos que los de 'GIL' y la fecha de alta será la fecha actual:

```
SQL> INSERT INTO EMPLE
  2  SELECT  1112, 'QUIROGA', OFICIO, DIR, SYSDATE, SALARIO,
  3  COMISION, DEPT_NO
  4  FROM EMPLE  WHERE APELLIDO='GIL';

1 fila creada.
```

Las columnas cuyos valores desconocemos (OFICIO, DIR, SALARIO, COMISION, DEPT_NO) son las que devolverá la sentencia SELECT; en el resto de las columnas ponemos directamente sus valores.

Actividades propuestas



2 **Dadas las tablas ALUM y NUEVOS, inserta en la tabla ALUM los nuevos alumnos.**

Inserta un empleado de apellido 'SAAVEDRA' con número 2000. La fecha de alta será la actual, el SALARIO será el mismo salario de 'SALA' más el 20 por 100 y el resto de datos serán los mismos que los datos de 'SALA'.

6.3 Modificación. Orden UPDATE

Para actualizar los valores de las columnas de una o varias filas de una tabla utilizamos la **orden UPDATE**, cuyo formato es el siguiente:

```
UPDATE     NombreTabla
SET        columna1=valor1, ..., columnan=valorn
WHERE      condición;
```

- *NombreTabla* es la tabla cuyas columnas se van a actualizar.
- *SET* indica las columnas que se van a actualizar y sus valores.
- *WHERE* selecciona las filas que se van a actualizar. Si se omite, la actualización afectará a todas las filas de la tabla.



Caso práctico

3 **Sea la tabla CENTROS, cambiamos la dirección del COD_CENTRO 22 a 'C/Pilón 13' y el número de plazas a 295:**

```
UPDATE CENTROS SET DIRECCION = 'C/Pilón 13', NUM_PLAZAS = 295 WHERE COD_CENTRO = 22;
```

¿Qué hubiese ocurrido si no hubiésemos puesto la cláusula WHERE?

```
SQL> UPDATE CENTROS SET DIRECCION = 'C/Pilón 13', NUM_PLAZAS = 295;
5 filas actualizadas.

SQL> SELECT * FROM CENTROS;
COD_CENTRO   T   NOMBRE                DIRECCION      TELEFONO    NUM_PLAZAS
----------   -   -------------------   ------------   ----------  ----------
        10   S   IES El Quijote        C/Pilón 13     965-887654         295
        15   P   CP Los Danzantes      C/Pilón 13     985-112322         295
        22   S   IES Planeta Tierra    C/Pilón 13     925-443400         295
        45   P   CP Manuel Hidalgo     C/Pilón 13     926-202310         295
        50   S   IES Antoñete          C/Pilón 13     989-406090         295
```

Como se aprecia, se hubieran modificado todas las filas de la tabla CENTROS con la dirección 'C/Pilón 13' y 295 en número de plazas.



Actividades propuestas

3 **Aumenta en 100 euros el salario y en 10 euros la comisión a todos los empleados del departamento 10, de la tabla EMPLE.**

A. UPDATE con SELECT

Podemos incluir una subconsulta en una sentencia UPDATE que puede estar contenida en la cláusula WHERE o puede formar parte de SET. Cuando la subconsulta (orden SELECT) forma parte de SET, debe seleccionar una única fila y el mismo número de columnas (con tipos de datos adecuados) que las que hay entre paréntesis al lado de SET. Los formatos son:

```
UPDATE  <NombreTabla>
SET     columna1 = valor1, columna2 = valor2, ...
WHERE   columna3 = (SELECT ...);
```




```
UPDATE    <NombreTabla>

SET  (columna1, columna2, ...) = (SELECT col1, col2, ...)

WHERE     condición;

UPDATE    <NombreTabla>

SET  columna1 = (SELECT col1 ...), columna2 = (SELECT
col2 ...)

WHERE     condición;
```

Caso práctico



4 **En la tabla CENTROS la siguiente orden UPDATE igualará la dirección y el número de plazas del código de centro 10 a los valores de las columnas correspondientes que están almacenadas para el código de centro 50. Los valores actuales de estos centros son:**

```
SQL> UPDATE CENTROS SET (DIRECCION, NUM_PLAZAS) = (SELECT DIRECCION,
     NUM_PLAZAS FROM CENTROS WHERE COD_CENTRO = 50) WHERE COD_CENTRO= 10;
```

A partir de la tabla EMPLE, cambia el salario a la mitad y la comisión a 0, a aquellos empleados que pertenezcan al departamento con mayor número de empleados.

```
SQL> UPDATE EMPLE SET SALARIO = SALARIO/2, COMISION = 0 WHERE DEPT_NO =
     (SELECT DEPT_NO FROM EMPLE GROUP BY DEPT_NO HAVING COUNT(*) =
     (SELECT MAX(COUNT(*)) FROM EMPLE GROUP BY DEPT_NO));
```

Para todos los empleados de la tabla EMPLE y del departamento de 'CONTABILIDAD', cambiamos su salario al doble del salario de 'SÁNCHEZ' y su apellido, a minúscula.

```
SQL> UPDATE  EMPLE SET APELLIDO = LOWER(APELLIDO),
     SALARIO = (SELECT SALARIO*2 FROM EMPLE WHERE APELLIDO = 'SANCHEZ')
     WHERE DEPT_NO = (SELECT DEPT_NO FROM DEPART
             WHERE DNOMBRE = 'CONTABILIDAD');
```

Actividades propuestas



4 **Modifica el número de departamento de 'SAAVEDRA'. El nuevo departamento será el departamento donde hay más empleados cuyo oficio sea 'EMPLEADO'.**

6.4 Borrado de filas. Orden DELETE

Para eliminar una fila o varias filas de una tabla se usa la **orden DELETE.** La cláusula WHERE es esencial para eliminar sólo aquellas filas deseadas. Sin la cláusula WHERE, DELETE borrará todas las filas de la tabla. El espacio usado por las filas que han sido borradas no se reutiliza, a menos que se realice un EXPORT o un IMPORT. La condición puede incluir una subconsulta. Éste es su formato:

```
DELETE [FROM] NombreTabla WHERE Condición;
```



Caso práctico

5 Borramos el COD_CENTRO 50 de la tabla CENTROS:

```
SQL> DELETE FROM CENTROS WHERE COD_CENTRO=50;
```

O bien:

```
SQL> DELETE CENTROS WHERE COD_CENTRO = 50;
```

Borramos todas las filas de la tabla CENTROS: `SQL> DELETE FROM CENTROS;`

Igualmente, podríamos haber puesto: `DELETE CENTROS;`

Borramos todas las filas de la tabla LIBRERIA cuyos EJEMPLARES no superen la media de ejemplares en su ESTANTE:

```
SQL> DELETE FROM LIBRERIA L WHERE EJEMPLARES <
     (SELECT AVG(EJEMPLARES) FROM LIBRERIA WHERE ESTANTE = L.ESTANTE
     GROUP BY ESTANTE);
```

Borramos los departamentos de la tabla DEPART con menos de cuatro empleados.

```
SQL> DELETE FROM DEPART WHERE DEPT_NO IN
      (SELECT DEPT_NO FROM EMPLE GROUP BY DEPT_NO HAVING COUNT(*) < 4);
```



Actividades propuestas

5 Borra de la tabla ALUM los ANTIGUOS alumnos.

Borra todos los departamentos de la tabla DEPART para los cuales no existan empleados en EMPLE.





6.5 ROLLBACK, COMMIT y AUTOCOMMIT

Supongamos que queremos borrar una fila de una tabla pero, al teclear la orden SQL, se nos olvida la cláusula WHERE y... ¡horror!, ¡borramos todas las filas de la tabla! Esto no es problema, pues Oracle permite dar marcha atrás a un trabajo realizado mediante la **orden ROLLBACK,** siempre y cuando no hayamos validado los cambios en la base de datos mediante la **orden COMMIT.**

Cuando hacemos transacciones sobre la base de datos, es decir, cuando insertamos, actualizamos y eliminamos datos en las tablas, los cambios no se aplicarán a la base de datos hasta que no hagamos un COMMIT. Esto significa que, si durante el tiempo que hemos estado realizando transacciones, no hemos hecho ningún COMMIT y de pronto se va la luz, todo el trabajo se habrá perdido, y nuestras tablas estarán en la situación de partida.

Una **transacción** es una secuencia de una o más sentencias SQL que juntas forman una unidad de trabajo.

Para validar los cambios que se hagan en la base de datos tenemos que ejecutar la orden COMMIT:

```
SQL> COMMIT;
Validación terminada.
```

SQL*Plus e iSQL*Plus permiten validar automáticamente las transacciones sin tener que indicarlo de forma explícita. Para eso sirve el **parámetro AUTOCOMMIT.** El valor de este parámetro se puede mostrar con la orden SHOW, de la siguiente manera:

```
SQL> SHOW AUTOCOMMIT;
autocommit OFF
```

OFF es el valor por omisión, de manera que las transacciones (INSERT, UPDATE y DELETE) no son definitivas hasta que no hagamos COMMIT. Si queremos que INSERT, UPDATE Y DELETE tengan un carácter definitivo sin necesidad de realizar la validación COMMIT, hemos de activar el parámetro AUTOCOMMIT con la **orden SET:**

```
SQL> SET AUTOCOMMIT ON;

SQL> SHOW  AUTOCOMMIT;
autocommit IMMEDIATE
```

Ahora, cualquier INSERT, UPDATE y DELETE se validará automáticamente.

La orden **ROLLBACK** aborta la transacción volviendo a la situación de las tablas de la base de datos desde el último COMMIT:

```
SQL>  ROLLBACK;
Rollback terminado.
```

La Figura 6.1 muestra transacciones típicas que ilustran las condiciones de COMMIT y ROLLBACK.

Figura 6.1. *Transacciones.*



Caso práctico

6 **Partimos de la tabla DEPART con todas sus filas. El usuario SCOTT abre una sesión desde SQL*Plus y ejecuta la orden: `SELECT * FROM  DEPART;` para consultar las filas de la tabla DEPART.**

- SCOTT abre otra sesión ahora desde iSQL*Plus, ejecuta la misma orden para ver el contenido de la tabla. Ambos ven las mismas filas.
- Desde SQL*Plus borra una fila de la tabla DEPART: `DELETE DEPART WHERE DEPT_NO=20;` y consulta de nuevo la tabla. Observa que hay una fila menos.

(Continúa)





(Continuación)

- Desde iSQL*Plus, consulta el contenido de la tabla DEPART. Observa que se muestran todas las filas.
- Desde SQL*Plus, ejecuta la orden: COMMIT; Todos los cambios realizados se validan en la base de datos.
- Desde iSQL*Plus, consulta otra vez el contenido de la tabla DEPART. Observa que ahora no se muestra la fila borrada desde SQL*Plus.
- Desde SQL*Plus, ejecuta la orden: `DELETE DEPART;` después consulta el contenido de la tabla, y observa que no se muestra ninguna fila.
- Se ha confundido al ejecutar la orden porque falta la claúsula WHERE en la sentencia DELETE, entonces ejecuta: `ROLLBACK;` De nuevo, consulta el contenido de la tabla: se muestran los datos desde el último COMMIT.

Actividades propuestas



6 **Práctica las órdenes ROLLBACK y COMMIT abriendo dos sesiones SQL con el mismo usuario y realiza transacciones sobre tus tablas.**

A. COMMIT implícito

Hay varias órdenes SQL que fuerzan a que se ejecute un COMMIT sin necesidad de indicarlo:

QUIT	DISCONNECT	CREATE VIEW	ALTER
EXIT	CREATE TABLE	DROP VIEW	REVOQUE
CONNECT	DROP TABLE	GRANT	AUDIT
			NOAUDIT

Usar cualquiera de estas órdenes es como usar COMMIT.

B. ROLLBACK automático

Si, después de haber realizado cambios en nuestras tablas, se produce un fallo del sistema (por ejemplo, se va la luz) y no hemos validado el trabajo, Oracle hace un ROLLBACK automático sobre cualquier trabajo no validado. Esto significa que tendremos que repetir el trabajo cuando pongamos en marcha la base de datos.

Conceptos básicos



A continuación se muestra un resumen de las órdenes vistas en el tema:

<table>
<tr><td>INSERT</td><td>Inserción de una fila:
<pre>
INSERT INTO NombreTabla [(columna [, columna] ...)]
VALUES (valor [, valor] ...);
</pre>
Inserción multifila:
<pre>
INSERT INTO NombreTabla1 [(columna [, columna] ...)]
        SELECT {columna [, columna] ... | *}
        FROM NombreTabla2 [CLÁUSULAS DE SELECT];
</pre></td></tr>
<tr><td>UPDATE</td><td>Modificación de filas:
<pre>
UPDATE <NombreTabla>
        SET   columna1 = valor1, ..., columnan = valorn
        WHERE condición;
</pre>
Modificación de filas con SELECT:
<pre>
UPDATE <NombreTabla>
SET   columna1 = valor1, columna2 = valor2, ...
WHERE   columna3=(SELECT ....);
UPDATE <NombreTabla>
SET   (columna1, columna2, ...)=(SELECT col1, col2, ...)
WHERE   condición;
UPDATE <NombreTabla>
SET   columna1 = (SELECT col1 ...), columna2 = (SELECT col2 ...)
WHERE   condición;
</pre></td></tr>
<tr><td>DELETE</td><td>Borrado de filas:
<pre>
DELETE [FROM] NombreTabla WHERE condición;
</pre></td></tr>
<tr><td>TRANSACCIONES</td><td>Validar los cambios:
<pre>
COMMIT;
</pre>
Abortar transacciones:
<pre>
ROLLBACK;
</pre></td></tr>
</table>





Actividades complementarias



Tablas PERSONAL, PROFESORES Y CENTROS

1 **Modifica el número de plazas con un valor igual a la mitad en aquellos centros con menos de dos profesores.**

2 **Elimina los centros que no tengan personal.**

3 **Añade un nuevo profesor en el centro o en los centros cuyo número de administrativos sea 1 en la especialidad de 'IDIOMA', con DNI 8790055 y de nombre 'Clara Salas'.**

4 **Borra al personal que esté en centros de menos de 300 plazas y con menos de dos profesores.**

5 **Borra a los profesores que estén en la tabla PROFESORES y que no estén en la tabla PERSONAL.**

Tablas ARTICULOS, FABRICANTES, TIENDAS, PEDIDOS Y VENTAS (hacer DESC de las tablas)

Un almacén de distribución de artículos desea mantener información sobre las ventas hechas por las tiendas que compran al almacén. Dispone de las siguientes tablas para mantener esta información:

ARTICULOS: almacena cada uno de los artículos que el almacén puede abastecer a las tiendas. Cada artículo viene determinado por las columnas: ARTICULO, COD_ FABRICANTE, PESO y CATEGORIA. La categoría puede ser 'Primera', 'Segunda' o 'Tercera'.

FABRICANTES: contiene los países de origen de los fabricantes de artículos. Cada COD_FABRICANTE tiene su país.

TIENDAS: almacena los datos de las tiendas que venden artículos. Cada tienda se identifica por su NIF.

PEDIDOS: son los pedidos que realizan las tiendas al almacén. Cada pedido se identifica por: NIF, ARTICULO, COD_FABRICANTE, PESO, CATEGORIA y FECHA_ PEDIDO. Cada fila de la tabla representa un pedido.

VENTAS: almacena las ventas de artículos que hace cada una de las tiendas. Cada venta se identifica por: NIF, ARTICULO, COD_ FABRICANTE, PESO, CATEGORIA y FECHA_ VENTA. Cada fila de la tabla representa una venta.

6 **Da de alta un nuevo artículo de 'Primera' categoría para los fabricantes de 'FRANCIA' y abastece con 5 unidades de ese artículo a todas las tiendas y en la fecha de hoy.**

7 **Inserta un pedido de 20 unidades en la tienda '1111-A' con el artículo que mayor número de ventas haya realizado.**

8 **Da de alta una tienda en la provincia de 'MADRID' y abastécela con 20 unidades de cada uno de los artículos existentes.**

9 **Da de alta dos tiendas en la provincia de 'SEVILLA' y abastécelas con 30 unidades de artículos de la marca de fabricante 'GALLO'.**

10 **Realiza una venta para todas las tiendas de 'TOLEDO' de 10 unidades en los artículos de 'Primera' categoría.**

11 **Para aquellos artículos de los que se hayan vendido más de 30 unidades, realiza un pedido de 10 unidades para la tienda con NIF '5555-B' con la fecha actual.**

12 **Cambia los datos de la tienda con NIF '1111-A' igualándolos a los de la tienda con NIF '2222-A'.**

13 **Cambia todos los artículos de 'Primera' categoría a 'Segunda' categoría del país 'ITALIA'.**

14 **Modifica aquellos pedidos en los que la cantidad pedida sea superior a las existencias del artículo, asignando el 20 por 100 de las existencias a la cantidad que se ha pedido.**

15 **Elimina aquellas tiendas que no han realizado ventas.**

16 **Elimina los artículos que no hayan tenido ni compras ni ventas.**

17 **Borra los pedidos de 'Primera' categoría cuyo país de procedencia sea 'BÉLGICA'.**

18 **Borra los pedidos que no tengan tienda.**

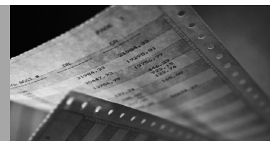
19 **Resta uno a las unidades de los últimos pedidos de la tienda con NIF '5555-B'.**

7

Creación, supresión y modificación de tablas, vistas y otros objetos

En esta unidad aprenderás a:

1. Manejar con fluidez las órdenes que permiten crear, modificar y suprimir tablas.
2. Usar con fluidez las órdenes que posibilitan crear y suprimir vistas y sinónimos.
3. Entender el concepto de integridad de datos.
4. Crear y modificar tablas con restricciones.
5. Descubrir la importancia que tiene emplear restricciones al crear tablas.
6. Descubrir las ventajas de recurrir a los sinónimos.
7. Utilizar sinónimos y vistas.




7.1 Introducción

Hasta el momento nos hemos dedicado a manipular tablas, es decir, a consultar datos de las tablas (SELECT), a insertar filas en ellas (INSERT), a eliminar filas (DELETE) y a modificar filas (UPDATE). Hemos llevado a cabo todas estas acciones con tablas que ya teníamos creadas, pero no hemos creado ninguna nueva.

En esta unidad empezaremos a usar el lenguaje de descripción de datos o DDL (*Data Description Language*). Manejaremos las órdenes CREATE, DROP y ALTER. La **orden CREATE** sirve para crear objetos de base de datos: tablas, vistas, sinónimos, etcétera. La **orden DROP** permite eliminar un objeto. Mediante la **orden ALTER** podemos modificar un objeto de base de datos.

Algo muy importante y útil son las restricciones. Hasta ahora, las tablas que hemos manipulado no presentaban muchas restricciones, únicamente NOT NULL, y ya hemos visto que, si se da un valor nulo a una columna que está definida como NOT NULL, se produce un error.

Otro ejemplo ilustrativo de la relevancia de las restricciones es el caso de que tengamos varias tablas: supongamos que disponemos de dos tablas, una de ARTÍCULOS y otra de VENTAS, y que intentamos introducir una venta de un artículo que no existe en la tabla ARTÍCULOS. Si no hemos definido restricciones en las tablas, se almacenará información incorrecta en la tabla VENTAS, con los consiguientes problemas para nuestro sistema de información.

7.2 Creación de una tabla

Empezaremos creando una tabla. Antes de hacerlo es conveniente planificar ciertos aspectos:

- **El nombre de la tabla.** Debe ser un nombre que identifique su contenido. Por ejemplo, llamamos a una tabla ALUMNOS porque contendrá datos sobre alumnos.
- **El nombre de cada columna de la tabla.** Ha de ser un nombre autodescriptivo, que identifique su contenido. Por ejemplo, DNI, NOMBRE o APELLIDOS.
- **El tipo de dato y el tamaño que tendrá cada columna.**
- **Las columnas obligatorias, los valores por defecto, las restricciones, etcétera.**

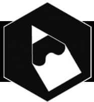
La denominación de la tabla puede tener de 1 a 30 caracteres de longitud. Ha de ser única y no puede ser una palabra reservada de Oracle. Su primer carácter debe ser alfabético y el resto pueden ser letras, números y el carácter de subrayado.

Para crear una tabla usamos la orden CREATE TABLE, cuyo formato más simple es:

```
CREATE  TABLE  Nombretabla
(
  Columna1   Tipo_dato  [NOT NULL],
  Columna2   Tipo_dato  [NOT NULL],
    ................
) [TABLESPACE espacio_de_tabla];
```

Donde:

- `Columna1, Columna2` son los nombres de las columnas que contendrá cada fila.
- `Tipo_dato` indica el tipo de dato (VARCHAR2, NUMBER, etcétera) de cada columna.
- `TABLESPACE espacio_de_tabla` señala el TABLESPACE para almacenar la tabla.
- `NOT NULL` indica que la columna debe contener alguna información; nunca puede ser nula.



Caso práctico

1 Creamos una tabla llamada ALUMNOS07:

```
CREATE TABLE ALUMNOS07
(
  NUMERO_MATRICULA      NUMBER(6)          NOT NULL,
  NOMBRE                VARCHAR2(15)       NOT NULL,
  FECHA_NACIMIENTO      DATE,
  DIRECCION             VARCHAR2(30),
  LOCALIDAD             VARCHAR2(15)
);
```

Esta sentencia crea una tabla de nombre ALUMNOS07 con cinco columnas llamadas: NUMERO_MATRICULA, NOMBRE, FECHA_NACIMIENTO, DIRECCION y LOCALIDAD. Los tipos de datos para cada columna son: NUMBER, VARCHAR2, DATE, VARCHAR2 y VARCHAR2, respectivamente. La longitud de cada columna es: 6 para el NUMERO_MATRICULA, 15 para el NOMBRE, 30 para la DIRECCION y 15 para la LOCALIDAD. Para el tipo de datos DATE no se define longitud. Oracle la asigna automáticamente.

En las columnas NUMERO_MATRICULA y NOMBRE se ha definido la restricción NOT NULL, indicándose que siempre deben tener algún valor al insertar una nueva fila. Dado que no se ha especificado la cláusula TABLESPACE, la tabla se almacenará en el *tablespace* que tenga asignado el usuario.

La ejecución de la sentencia daría la siguiente salida: `Tabla creada`.

(Continúa)



(Continuación)

Hemos de hacer algunas observaciones:

- Las definiciones individuales de columnas se separan mediante comas.
- No se pone coma después de la última definición de columna.
- Las mayúsculas y minúsculas son indiferentes a la hora de crear una tabla.

Si intentamos crear la tabla ALUMNOS07 y existe otra tabla con este nombre, aparecerá un mensaje de error. Si volvemos a escribir las órdenes anteriores nos dará el siguiente error:

```
CREATE TABLE ALUMNOS07
            *
ERROR en línea 1:
ORA-00955: este nombre ya lo está utilizando otro objeto existente
```

Los usuarios pueden consultar las tablas creadas por medio de la vista **USER_TABLES.** Esta vista contiene información acerca de las tablas: nombre de la tabla, nombre del *tablespace*, número de filas, información de almacenamiento, etcétera. Por ejemplo, si queremos visualizar el nombre de las tablas creadas, tendríamos que escribir la orden:

```
SELECT TABLE_NAME FROM USER_TABLES;
```

Existen otras dos vistas que permiten obtener información de los objetos que son propiedad del usuario:

- **USER_OBJECTS:** objetos que son propiedad del usuario.
- **USER_CATALOG:** tablas, vistas, sinónimos y secuencias propiedad del usuario.

Actividades propuestas



1 **Investiga las columnas de USER_TABLES y utilízalas para consultar el nombre de las tablas que tienes y el número de filas que tiene cada tabla.**

Investiga las columnas de USER_OBJECTS y USER_CATALOG y utiliza las vistas para consultar las tablas que tienes.

A. Integridad de datos

Cuando almacenamos datos en nuestras tablas, se ajustan a una serie de restricciones predefinidas. Por ejemplo, que una columna no pueda almacenar valores negativos, que una cadena de caracteres se deba almacenar en mayúsculas o que una columna no pueda ser 0. La **integridad** hace referencia al hecho de que los datos de la base de datos han de ajustarse a restricciones antes de almacenarse en ella. Así pues, una **restricción de integridad** será una regla que restringe el rango de valores para una o más columnas en la tabla.

Si se produce cualquier fallo mientras un usuario está cambiando los datos en la base de datos, ésta tiene la capacidad de deshacer o cancelar cualquier transacción sospechosa.

El **objetivo de las restricciones** es que las aplicaciones o los usuarios que van a manipular los datos tengan menos trabajo, y que sea Oracle el que realice la mayor parte de las tareas de mantenimiento de la integridad de la base de datos.

Existe otro tipo de integridad, que es la **integridad referencial,** la cual garantiza que los valores de una columna (o columnas) de una tabla (*clave ajena*) dependan de los valores de otra columna (o columnas) de otra tabla (*clave primaria*). Si en el ejemplo comentado al principio de la unidad sobre las tablas VENTAS y ARTÍCULOS, se define integridad referencial para estas tablas, nunca se dará la situación de insertar una venta con un artículo inexistente. Todas estas restricciones de integridad se explican en el siguiente apartado.

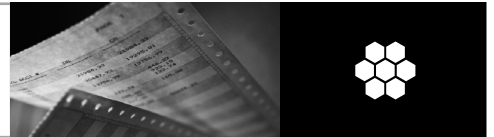
B. Restricciones en CREATE TABLE

La orden CREATE TABLE permite definir distintos tipos de restricciones sobre una tabla: claves primarias, claves ajenas, obligatoriedad, valores por defecto y verificación de condiciones.

Para definir las restricciones en la orden CREATE TABLE usamos la **cláusula CONSTRAINT.** Ésta puede restringir una sola columna (*restricción de columna*) o un grupo de columnas de una misma tabla (*restricción de tabla*). Hay dos modos de especificar restricciones: como parte de la definición de columnas (una restricción de columna) o al final, una vez especificadas todas las columnas (una restricción de tabla).

A continuación, aparece el formato de la orden CREATE TABLE con restricción de columna. La restricción forma parte de la definición de la columna:

```
CREATE  TABLE  nombre_tabla
(
  Columna1  TIPO_DE_DATO
    [CONSTRAINT nombrerestricción]
    [NOT NULL] [UNIQUE] [PRIMARY KEY] [DEFAULT valor]
    [REFERENCES Nombretabla [(columna [, columna])]
              [ON DELETE CASCADE]]
    [CHECK (condición)],
  Columna2  TIPO_DE_DATO
    [CONSTRAINT nombrerestricción]
    [NOT NULL] [UNIQUE] [PRIMARY KEY] [DEFAULT valor]
    [REFERENCES Nombretabla [(columna [, columna])]
              [ON DELETE CASCADE]]
    [CHECK (condición)],
  ...
) [TABLESPACE espacio_de_tabla];
```



Ejemplo:

```
CREATE TABLE EMPLEADO
(
NOMBRE         VARCHAR2(25) PRIMARY KEY,
EDAD           NUMBER       CHECK (EDAD BETWEEN 18 AND 35),
COD_PROVINCIA  NUMBER(2)    REFERENCES PROVINCIAS ON DELETE
                            CASCADE
);
```

Aquí se definen las siguientes restricciones:

- Clave primaria: NOMBRE.
- Clave ajena: COD_PROVINCIA, que referencia a la tabla PROVINCIAS.
- Verificación de condición CHECK: la edad ha de estar comprendida entre 18 y 35.

Las restricciones de la orden CREATE TABLE que aparecen al final de la definición de las columnas (o de tabla) se diferencian de la anterior en que se puede hacer referencia a varias columnas en una única restricción (por ejemplo, declarando dos columnas como clave primaria o ajena):

```
CREATE  TABLE  nombre_tabla
(
 Columna1   TIPO_DE_DATO [NOT NULL],
 Columna2   TIPO_DE_DATO [NOT NULL],
 Columna3   TIPO_DE_DATO [NOT NULL],
 ...
[CONSTRAINT nombrerestricción]
     {[UNIQUE] | [PRIMARY KEY] (columna[,columna])},
[CONSTRAINT nombrerestricción]
     [FOREIGN KEY  (columna[,columna])
       REFERENCES Nombretabla [(columna[,columna])]
       [ON DELETE CASCADE]],
[CONSTRAINT nombrerestricción]
      [CHECK (condición)],
  ...
) [TABLESPACE espacio_de_tabla];
```

Caso práctico



2 **Creamos la tabla PROVIN y la tabla EMPLEADO. Primero creamos PROVIN ya que EMPLEADO hace referencia a dicha tabla:**

```
CREATE TABLE PROVIN
(
 CODIGO  NUMBER(2) PRIMARY KEY,
 NOMBRE  VARCHAR2(25)
);
```

(Continúa)

(Continuación)

```
CREATE TABLE EMPLEADO
(
NOMBRE              VARCHAR2(25),
EDAD                NUMBER,
COD_PROVINCIA       NUMBER(2),
CONSTRAINT          PK_EMPLEADO PRIMARY KEY (NOMBRE),
CONSTRAINT          CK_EDAD    CHECK(EDAD BETWEEN  18 AND  35),
CONSTRAINT          FK_EMPLEADO FOREIGN KEY  (COD_PROVINCIA)
                    REFERENCES PROVIN ON DELETE  CASCADE
);
```

El nombre de las restricciones es opcional. También es válida esta sentencia CREATE TABLE:

```
CREATE TABLE EMPLEADO
(
NOMBRE              VARCHAR2(25),
EDAD                NUMBER(2),
COD_PROVINCIA       NUMBER(2),
PRIMARY KEY (NOMBRE),
CHECK (EDAD BETWEEN 18 AND 35),
FOREIGN KEY (COD_PROVINCIA) REFERENCES PROVIN
                             ON DELETE CASCADE
);
```

Clave primaria. La restricción PRIMARY KEY

Una **clave primaria dentro de una tabla** es una columna o un conjunto de columnas que identifican unívocamente a cada fila. Debe ser única, no nula y obligatoria. Como máximo podemos definir una clave primaria por tabla. Esta clave se puede referenciar por una columna o columnas de otra tabla. Llamamos clave ajena a esta columna o columnas. Cuando se crea una clave primaria, automáticamente se crea un índice que facilita el acceso a la tabla. Para definir una clave primaria en una tabla usamos la restricción **PRIMARY KEY.**

Éstos son los formatos de la orden CREATE TABLE para definir claves primarias:

– El **formato de restricción de columna** es el siguiente:

```
CREATE TABLE nombre_tabla
(
  Columna1 TIPO_DE_DATO [CONSTRAINT nombrerestricción]
                             PRIMARY KEY,
  Columna2  TIPO_DE_DATO,
  ...
) [TABLESPACE espacio_de_tabla];
```



– Y el **formato de restricción de tabla** es éste:

```
CREATE TABLE nombre_tabla
(
  Columna1   TIPO_DE_DATO,
  Columna2   TIPO_DE_DATO,
  ...
  [CONSTRAINT nombrerestricción] PRIMARY KEY (columna [,
                                  columna]),
  ...
) [TABLESPACE espacio_de_tabla];
```

Caso práctico

3 **Creamos la tabla BLOQUESPISOS. Las columnas son las siguientes:**

Nombre columna	Representa	Tipo
CALLE	Calle donde está el bloque	VARCHAR2(30)
NUMERO	Número donde está el bloque	NUMBER(3)
PISO	Número de planta	NUMBER(2)
PUERTA	Puerta	CHAR(1)
CODIGO_POSTAL	Código postal	NUMBER(5)
METROS	Metros de la vivienda	NUMBER(5)
COMENTARIOS	Otros datos de la vivienda	VARCHAR2(60)
COD_ZONA	Código de zona donde está el bloque	NUMBER(2)
DNI	DNI del propietario	VARCHAR2(10)

La clave primaria estará formada por las columnas CALLE, NUMERO, PISO y PUERTA que, por tanto, no pueden contener valores nulos. Se puede crear la tabla de la siguiente forma:

```
CREATE TABLE BLOQUESPISOS
(
 CALLE              VARCHAR2(30)       NOT NULL,
 NUMERO             NUMBER(3)          NOT NULL,
 PISO               NUMBER(2)          NOT NULL,
 PUERTA             CHAR(1)            NOT NULL,
 CODIGO_POSTAL      NUMBER(5),
 METROS             NUMBER(5),
 COMENTARIOS        VARCHAR2(60),
 COD_ZONA           NUMBER(2),
 DNI                VARCHAR2(10),
 CONSTRAINT PK_VIV PRIMARY KEY (CALLE, NUMERO, PISO, PUERTA)
);
```

La última sentencia se podría haber puesto de la siguiente manera: **`PRIMARY KEY (CALLE, NUMERO, PISO, PUERTA`**), sin dar nombre a la restricción.

(Continúa)

(Continuación)

La clave de esta tabla es la combinación de CALLE, NUMERO, PISO y PUERTA, que están especificadas como NOT NULL. Esto permite evitar la introducción de datos en la tabla sin dar valores a determinadas columnas. Si no ponemos NOT NULL en alguno de los atributos de la clave, Oracle automáticamente coloca NOT NULL en ese atributo. Al visualizar la descripción de la tabla comprobamos que estas cuatro columnas tienen la restricción NOT NULL.

Creamos la tabla ZONAS. Las columnas para esta tabla son:

Nombre columna	Representa	Tipo
COD_ZONA	Código de la zona	NUMBER(2)
NOMBREZONA	Nombre de zona	VARCHAR2(20)
MASDATOS	Otros datos de la zona	VARCHAR2(50)

La clave primaria es el código de zona (COD_ZONA) y la definimos formando parte de la columna (restricción de columna):

```
CREATE TABLE ZONAS
(
        COD_ZONA       NUMBER(2)          PRIMARY KEY,
        NOMBREZONA     VARCHAR2(15)       NOT NULL,
        MASDATOS       VARCHAR2(60)
);
```

Si en una tabla forman parte de la clave primaria varias columnas, ésta no se puede definir como restricción de columna. Cuando en la orden CREATE TABLE aparece la cláusula PRIMARY KEY sólo se debe especificar una vez.

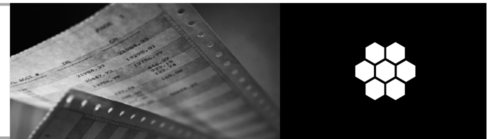


Actividades propuestas

2 **Prueba a crear la tabla BLOQUESPISOS definiendo la restricción de clave primaria como restricción de columna. Comenta el error que aparece.**

Claves ajenas. La restricción FOREIGN KEY

Una clave ajena está formada por una o varias columnas que están asociadas a una clave primaria de otra o de la misma tabla. Se pueden definir tantas claves ajenas como sea preciso, y pueden estar o no en la misma tabla que la clave primaria. El valor de la columna o columnas que son claves ajenas debe ser NULL o igual a un valor de la clave referenciada (regla de integridad referencial).



El formato de CREATE TABLE para definir claves ajenas puede ser cualquiera de los siguientes:

- **Formato de restricción de columna.** La clave ajena se define en la descripción de la columna usando la cláusula REFERENCES:

```
CREATE TABLE  nombre_tabla
(
  Columna1  TIPO_DE_DATO
    [CONSTRAINT nombrerestricción]
    REFERENCES Nombretabla [(columna)] [ON DELETE CASCADE],
  ...
  Columna2  TIPO_DE_DATO, ...
) [TABLESPACE espacio_de_tabla];
```

- **Formato de restricción de tabla.** La clave ajena se define al final de todas las columnas empleando las cláusulas FOREIGN KEY y REFERENCES:

```
CREATE TABLE  nombre_tabla
(
      Columna1   TIPO_DE_DATO,
      Columna2   TIPO_DE_DATO,
      ...
      [CONSTRAINT nombrerestricción] FOREIGN KEY (columna
           [, columna]) REFERENCES Nombretabla [(columna[,
           columna])] [ON DELETE CASCADE],
      ...
    ) [TABLESPACE espacio_de_tabla];
```

En la cláusula **REFERENCES** indicamos la tabla a la cual remite la clave ajena. La derecha de FOREIGN KEY y, entre paréntesis, indicamos la columna o columnas que forman parte de la clave ajena.

La cláusula **ON DELETE CASCADE** o **borrado en cascada** se define cuando al borrar las filas asociadas con claves primarias deseamos que se eliminen automáticamente las filas con claves ajenas que referencien a dichas claves.

Caso práctico



4 **Sean las tablas PERSONAS y PROVINCIAS. La tabla PERSONAS contiene datos sobre las personas de una comunidad, mientras que la tabla PROVINCIAS contiene el código y nombre de cada provincia, se relacionan por el atributo COD_PROVIN:**

(Continúa)

(Continuación)

TABLA PERSONAS:	TABLA PROVINCIAS:
DNI	**CODPROVINCIA**
NOMBRE	NOM_PROVINCIA
DIRECCION	
POBLACION	
CODPROVIN	

Donde:

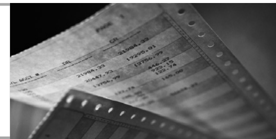
- DNI es la clave primaria de la tabla PERSONAS.
- CODPROVINCIA de la tabla PROVINCIAS es clave primaria de esta tabla.
- CODPROVIN de la tabla PERSONAS es clave ajena, porque se relaciona con la clave primaria de la tabla PROVINCIAS. Los valores que se almacenen en esta columna deben coincidir con la clave primaria de la tabla PROVINCIAS. Se puede afirmar que PROVINCIAS es la tabla maestra y PERSONAS es la tabla detalle.

Hemos de crear, en primer lugar, la tabla PROVINCIAS y, después, la tabla PERSONAS, ya que PERSONAS referencia a PROVINCIAS. Si creamos primero la tabla PERSONAS y la tabla PROVINCIAS no está creada, Oracle emitirá un mensaje de error.

```
CREATE TABLE PROVINCIAS
(
   CODPROVINCIA   NUMBER(2)      PRIMARY KEY,
   NOM_PROVINCIA  VARCHAR2(15)
);
CREATE TABLE PERSONAS
(
   DNI            NUMBER(8)      PRIMARY KEY,
   NOMBRE         VARCHAR2(15),
   DIRECCION      VARCHAR2(25),
   POBLACION      VARCHAR2(20),
   CODPROVIN      NUMBER(2) NOT NULL REFERENCES PROVINCIAS
);
```

La clave ajena se ha definido usando la cláusula REFERENCES, aunque también se puede definir usando la cláusula FOREIGN KEY de la siguiente manera:

```
CREATE TABLE PERSONAS
(
   DNI           NUMBER(8)       PRIMARY KEY,
   NOMBRE        VARCHAR2(15),
   DIRECCION     VARCHAR2(25),
   POBLACION     VARCHAR2(20),
   CODPROVIN     NUMBER(2)       NOT NULL,
   FOREIG KEY   (CODPROVIN) REFERENCES PROVINCIAS
);
```





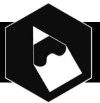
Actividades propuestas



3 Hemos creado las tablas PROVINCIAS y PERSONAS. Inserta filas en las tablas.

Inserta filas en la tabla PERSONAS dando a CODPROVIN un valor que no exista en PROVINCIAS. ¿Qué ocurre? Comenta el resultado.

Caso práctico



5 Partimos de una situación en que las dos tablas tienen datos. Vamos a borrar todas las filas de la tabla PROVINCIAS:

```
DELETE PROVINCIAS;
*
ERROR en línea 1:
ORA-02292: restricción de integridad (SCOTT.SYS_C005389) violada - registro secun-
dario encontrado
```

Se produce un error: no podemos borrar filas en la tabla maestra (PROVINCIAS) si hay filas en la tabla detalle (PERSONAS) que las estén referenciando. Es decir, una fila no se puede borrar si es referenciada por alguna clave ajena.

Si se añade la cláusula ON DELETE CASCADE en la opción REFERENCES de la tabla detalle (PERSONAS) se podrán eliminar las filas de la tabla maestra (PROVINCIAS) y las filas correspondientes en la tabla detalle (PERSONAS) con esa provincia serán eliminadas. Borramos la tabla PERSONAS de la siguiente manera: `DROP TABLE PERSONAS;` y la volvemos a crear así:

```
CREATE TABLE PERSONAS
(
   DNI             NUMBER(8)          PRIMARY KEY,
   NOMBRE          VARCHAR2(15),
   DIRECCION       VARCHAR2(25),
   POBLACION       VARCHAR2(20),
   CODPROVIN       NUMBER(2)          NOT NULL,
   FOREIGN KEY     (CODPROVIN) REFERENCES PROVINCIAS ON DELETE CASCADE
);
```

Como veremos más adelante, DROP TABLE se utiliza para suprimir una tabla de la base de datos.

Una vez creada la tabla insertamos filas y borramos una fila de la tabla maestra (PROVINCIAS). Automáticamente se borrarán las filas de la tabla detalle que se correspondan con las filas de la tabla maestra. Esta acción mantiene automáticamente la integridad referencial.

A la hora de borrar tablas relacionadas, primero se ha de borrar la tabla detalle y después la tabla maestra. Si se borra la tabla maestra antes que la tabla detalle se producirá un error:

```
DROP TABLE PROVINCIAS;
          *
ERROR en línea 1:
ORA-02449: claves únicas/primarias en la tabla referidas
por claves ajenas
```

De esta manera, se indica que hay claves ajenas que hacen referencia a esta tabla.

Hasta ahora, al definir las restricciones de clave primaria y clave ajena, no les hemos dado nombre. Por defecto, Oracle asigna un nombre a la restricción SYS_C00n, donde '*n*' es un número asignado automáticamente por Oracle. En el siguiente ejemplo se intenta insertar una fila en la tabla PROVINCIAS cuyo código de provincia (clave primaria) ya existe:

```
INSERT INTO PROVINCIAS VALUES(6,'CÁCERES');
*
ERROR en línea 1:
ORA-00001: restricción única (SCOTT.SYS_C005386) violada
```

Oracle da un mensaje de error: la restricción que ha sido violada es la SYS_C005386, que hace referencia a la violación de la clave primaria. El propietario de la tabla es el usuario llamado SCOTT.

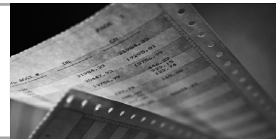
En el siguiente ejemplo se pretende insertar una fila en la tabla PERSONAS cuyo código de provincia (clave ajena) no existe en la tabla PROVINCIAS:

```
INSERT INTO PERSONAS VALUES(1122, 'Pedro', 'La Peña 16',
'Berrocalejo', 25);
*
ERROR en línea 1:
ORA-02291: restricción de integridad (SCOTT.SYS_C005392)
violada - clave principal no encontrada
```

La restricción que se ha violado ahora es la SYS_C005392, que hace referencia a la inexistencia del valor de la clave primaria en la tabla PROVINCIAS al que remite la clave ajena.

El nombre de una restricción es un nombre único que define el propietario del objeto o, por defecto, el sistema. Se asigna en el momento de definir la restricción. Por defecto, la denominación es SYS_C00n. La cláusula que da nombre a la restricción es la siguiente:

CONSTRAINT *nombrerestricción.*




Caso práctico



6 **Borramos las tablas PERSONAS y PROVINCIAS: `DROP TABLE PERSONAS; DROP TABLE PROVINCIAS;` y creamos las tablas de nuevo dando nombre a las restricciones de clave primaria y clave ajena:**

```
CREATE TABLE PROVINCIAS
(
    CODPROVINCIA    NUMBER(2)      CONSTRAINT PK_PROV PRIMARY KEY,
    NOM_PROVINCIA   VARCHAR2(15)
);
```

PK_PROV es el nombre de la restricción de clave primaria.

```
CREATE TABLE PERSONAS
(
    DNI             NUMBER(8)      CONSTRAINT PK_PER PRIMARY KEY,
    NOMBRE          VARCHAR2(15),
    DIRECCION       VARCHAR2(25),
    POBLACION       VARCHAR2(20),
    CODPROVIN       NUMBER(2) NOT NULL,
    CONSTRAINT FK_PER FOREIGN KEY (CODPROVIN) REFERENCES PROVINCIAS
                    ON DELETE  CASCADE
);
```

PK_PER es el nombre de la restricción de clave primaria. FK_PER es el nombre de la restricción de clave ajena.

A continuación se insertarán algunas filas en la tabla PROVINCIAS para hacer que se disparen las restricciones:

```
INSERT INTO PROVINCIAS VALUES(28, 'MADRID');
INSERT INTO PROVINCIAS VALUES(28,'SEVILLA');
*
ERROR en línea 1:
ORA-00001: restricción única (SCOTT.PK_PROV) violada
```

Resulta más fácil identificar una violación de una restricción si se le da un nombre al definirla.

Al insertar un código de provincia existente se produce un error en la restricción de clave primaria llamada PK_PROV. Ahora insertamos una fila en la tabla PERSONAS cuyo código de provincia no exista en la tabla PROVINCIAS:

```
INSERT INTO PERSONAS VALUES(1133, 'Luis', 'La Peña 12',  'Berrocalejo', 22);

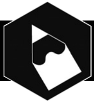
ERROR en línea 1:
ORA-02291: restricción de integridad (SCOTT.FK_PER) violada - clave principal no
encontrada
```

Se produce un error en la restricción de clave ajena llamada FK_PER.

Obligatoriedad. La restricción NOT NULL

Esta restricción asociada a una columna significa que no puede tener valores nulos, es decir, que ha de tener obligatoriamente un valor. En caso contrario, causa una excepción. En ejemplos anteriores nos hemos ocupado de cómo se define una columna con la restricción NOT NULL. Éste es su formato:

```
CREATE  TABLE  nombre_tabla
(
  Columna1  TIPO_DE_DATO [CONSTRAINT nombrerestricción]
                         NOT NULL,
  Columna2  TIPO_DE_DATO
  ...
) [TABLESPACE espacio_de_tabla];
```



Caso práctico

7 **Creamos una tabla definiendo las columnas NIF y NOMBRE como no nulas y damos nombre a la restricción no nula de la columna NOMBRE:**

```
CREATE TABLE EJEMPLO
(
  NIF       VARCHAR2(10)   NOT NULL,
  NOMBRE    VARCHAR(30)    CONSTRAINT NOMNONULO NOT NULL,
  EDAD      NUMBER(2)
);

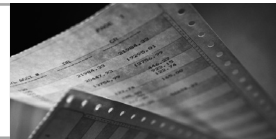
INSERT INTO EJEMPLO (NIF) VALUES ('45222111-A');
*
ERROR en línea 1:
ORA-01400: no se puede realizar una inserción NULL en
("SCOTT"."EJEMPLO"."NOMBRE")
```

Cuando se viola una columna NOT NULL se produce una excepción y, aunque se dé nombre a una restricción NOT NULL, no aparecerán los mensajes de restricción vistos antes, en los que aparecía el nombre de la restricción violada.

Valores por defecto. La especificación DEFAULT

En el momento de crear una tabla podemos asignar valores por defecto a las columnas. Si especificamos la cláusula DEFAULT a una columna, le proporcionamos un valor por omisión cuando el valor de la columna no se especifica en la cláusula INSERT.

En la especificación DEFAULT es posible incluir varias expresiones: constantes, funciones SQL y variables UID y SYSDATE. No se puede hacer referencias a columnas o a funciones PL/SQL.




Caso práctico



8 **Se crea la tabla EJEMPLO1 y se asigna a la columna FECHA la fecha del sistema:**

```
CREATE TABLE EJEMPLO1
(
  DNI         VARCHAR2(10) NOT NULL,
  NOMBRE      VARCHAR(30),
  FECHA       DATE DEFAULT SYSDATE
);
```

Se inserta una fila en la tabla dando valores a todas las columnas, salvo a la columna FECHA: `INSERT INTO EJEMPLO1(DNI, NOMBRE) VALUES('1234','PEPA');` Al visualizar el contenido de la tabla, en la columna FECHA se almacenará la fecha del sistema, ya que no se dio valor.

Actividades propuestas



4 **Crea la tabla EJEMPLO2 con las columnas DNI, NOMBRE y USUARIO; y asigna por defecto a la columna NOMBRE el literal 'No definido' y a la columna USUARIO, el número identificativo del usuario (pseudocolumna UID). Inserta una fila en la tabla dando valor solo al DNI y visualiza el contenido.**

Verificación de condiciones. La restricción CHECK

Muchas columnas de tablas requieren valores limitados dentro de un rango o el cumplimiento de ciertas condiciones. Con una restricción de verificación de condiciones se puede expresar una condición que ha de cumplirse para todas y cada una de las filas de la tabla.

La **restricción CHECK** actúa como una cláusula WHERE. Puede hacer referencia a una o a más columnas, pero no a valores de otras filas. En una cláusula CHECK no cabe incluir subconsultas ni las pseudocolumnas SYSDATE, UID y USER.

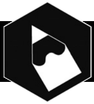
Éstos son los formatos con la orden CREATE TABLE con la restricción CHECK:

– Formato de restricción de columna:

```
CREATE TABLE  nombre_tabla
(
  Columna1 TIPO_DE_DATO [CONSTRAINT nombrerestricción]
                         CHECK (condición),
  Columna2 TIPO_DE_DATO
  ...
) [TABLESPACE espacio_de_tabla];
```

– Formato de restricción de tabla:

```
CREATE TABLE  nombre_tabla
(
  Columna1   TIPO_DE_DATO,
  Columna2   TIPO_DE_DATO,
  ...
  [CONSTRAINT nombrerestricción] CHECK (condición),
  ...
) [TABLESPACE espacio_de_tabla];
```



Caso práctico

9 **Se crea la tabla EJEMPLO3. Las columnas son: DNI VARCHAR2(10), NOMBRE VARCHAR2(30), EDAD NUMBER(2), CURSO NUMBER; y las restricciones:**

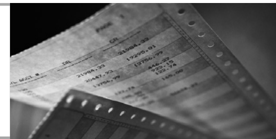
- El DNI no puede ser nulo.
- La clave primaria es el DNI.
- El NOMBRE no puede ser nulo.
- La EDAD ha de estar comprendida entre 5 y 20 años.
- El NOMBRE ha de estar en mayúsculas.
- El CURSO sólo puede almacenar 1, 2 ó 3.

Es posible crear la tabla de varias maneras: se puede dar nombre o no a las restricciones, definir las restricciones en la descripción de la columna o al final, combinando ambas, etcétera:

```
CREATE TABLE EJEMPLO3
(
  DNI           VARCHAR2(10)     NOT NULL,
  NOMBRE        VARCHAR2(30)     NOT NULL,
  EDAD          NUMBER(2),
  CURSO         NUMBER,
  CONSTRAINT    CLAVE_P          PRIMARY KEY(DNI),
  CONSTRAINT    COMP_EDAD        CHECK (EDAD BETWEEN 5 AND 20),
  CONSTRAINT    NOMBRE_MAYUS     CHECK (NOMBRE = UPPER(NOMBRE)),
  CONSTRAINT    COMP_CURSO       CHECK (CURSO IN(1, 2, 3))
);
```

Sin dar nombre a las restricciones y definiéndolas en la descripción de las columnas, tenemos:

```
CREATE TABLE EJEMPLO3
(
  DNI           VARCHAR2(10)     NOT NULL PRIMARY KEY,
  NOMBRE        VARCHAR2(30)     NOT NULL CHECK (NOMBRE = UPPER(NOMBRE)),
  EDAD          NUMBER(2)        CHECK (EDAD BETWEEN 5 AND 20),
  CURSO         NUMBER           CHECK (CURSO IN(1, 2, 3))
);
```




Actividades propuestas



5 **Inserta filas en la tabla anterior haciendo fallar las restricciones.**

La restricción NOT NULL es similar a: **CHECK**(*nombre_columna* **IS NOT NULL**).

Actividades propuestas



6 **Crea las siguientes tablas con las restricciones definidas.**

Tabla FABRICANTES

COD_FABRICANTE	NUMBER(3)
NOMBRE	VARCHAR2(15)
PAIS	VARCHAR2(15)

Tabla ARTICULOS

ARTICULO	VARCHAR2(20)
COD_FABRICANTE	NUMBER(3)
PESO	NUMBER(3)
CATEGORIA	VARCHAR2(10)
PRECIO_VENTA	NUMBER(6,2)
PRECIO_COSTO	NUMBER(6,2)
EXISTENCIAS	NUMBER(5)

Restricciones para la tabla FABRICANTES:

– La clave primaria es COD_FABRICANTE.

– Las columnas NOMBRE y PAIS han de almacenarse en mayúscula.

Restricciones para la tabla ARTICULOS:

– La clave primaria está formada por las columnas: ARTICULO, COD_FABRICANTE, PESO y CATEGORIA.

– COD_FABRICANTE es clave ajena que referencia a la tabla FABRICANTES.

– PRECIO_VENTA, PRECIO_COSTO y PESO han de ser > 0.

– CATEGORIA ha de ser 'Primera', 'Segunda' o 'Tercera'.

La restricción UNIQUE

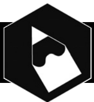
Evita valores repetidos en la misma columna. Puede contener una o varias columnas. Es similar a la restricción PRIMARY KEY, salvo que son posibles varias columnas UNIQUE definidas en una tabla. Admite valores NULL. Al igual que en PRIMARY KEY, cuando se define una restricción UNIQUE se crea un índice automáticamente. Veamos su formato:

– Formato de restricción de columna:

```
CREATE  TABLE  nombre_tabla
(
   Columna1 TIPO_DE_DATO [CONSTRAINT nombrerestricción]
                          UNIQUE,
   Columna2  TIPO_DE_DATO
   ...
 ) [TABLESPACE espacio_de_tabla];
```

– Formato de restricción de tabla:

```
CREATE  TABLE  nombre_tabla
(
   Columna1   TIPO_DE_DATO,
   Columna2   TIPO_DE_DATO,
   ...
   [CONSTRAINT nombrerestricción] UNIQUE (columna [,
                                  columna]),
   ...
 ) [TABLESPACE espacio_de_tabla];
```



Caso práctico

10 **Se crea la tabla EJEMPLO1_U definiendo una columna con UNIQUE y se intentan insertar dos filas, una de ellas violando la restricción.**

```
CREATE TABLE EJEMPLO1_U
(
  DNI    VARCHAR2(10)     PRIMARY KEY,
  NOM    VARCHAR2(30)     UNIQUE,
  EDAD   NUMBER(2)
);
```

Se inserta la primera fila:

```
INSERT INTO EJEMPLO1_U VALUES ('11111', 'PEPA', 20);
```

(Continúa)



(Continuación)

Se inserta la segunda fila:

```
INSERT INTO EJEMPLO1_U  VALUES ('11112', 'PEPA', 21);
*
ERROR en línea 1:
ORA-00001: restricción única (SCOTT.SYS_C005401) violada
```

Vistas del diccionario de datos para las restricciones

Existe una serie de vistas creadas por Oracle que contienen información referente a las restricciones definidas en las tablas. Contienen información general las siguientes:

- USER_CONSTRAINTS: definiciones de restricciones de tablas propiedad del usuario.
- ALL_CONSTRAINTS: definiciones de restricciones sobre tablas a las que puede acceder el usuario.
- DBA_CONSTRAINTS: todas las definiciones de restricciones sobre todas las tablas.

El tipo de restricción, columna CONSTRAINT_TYPE de estas vistas, puede ser:

C Restricciones de tipo CHECK	R Restricción FOREIGN KEY (References)
P Restricción PRIMARY KEY	U Restricción UNIQUE

Para información sobre restricciones en las columnas tenemos:

- USER_CONS_COLUMNS: información sobre las restricciones de columnas en tablas del usuario.
- ALL_CONS_COLUMNS: información sobre las restricciones de columnas en tablas a las que puede acceder el usuario.
- DBA_CONS_COLUMNS: información sobre todas las restricciones de columnas.

Caso práctico



11 **Observemos las restricciones de la tabla EJEMPLO: el nombre de restricción, el nombre de la tabla y el tipo de restricción: `SELECT CONSTRAINT_NAME, TABLE_NAME, CONSTRAINT_TYPE FROM USER_CONSTRAINTS WHERE TABLE_NAME = 'EJEMPLO';`**

Para ver las restricciones definidas en las columnas de la tabla 'EJEMPLO' se teclea la orden siguiente: `SELECT CONSTRAINT_NAME, TABLE_NAME, COLUMN_NAME FROM USER_CONS_COLUMNS WHERE TABLE_NAME = 'EJEMPLO';`

C. Creación de una tabla con datos recuperados en una consulta

La sentencia CREATE TABLE permite crear una tabla a partir de la consulta de otra tabla ya existente. La nueva tabla contendrá los datos obtenidos en la consulta. Se lleva a cabo esta acción con la cláusula AS colocada al final de la orden CREATE TABLE. El formato es el que sigue:

```
CREATE TABLE Nombretabla
(
  Columna [, Columna]
)
[TABLESPACE espacio_de_tabla]
AS consulta;
```

No es necesario especificar tipos ni tamaño de las columnas, ya que vienen determinados por los tipos y los tamaños de las recuperadas en la consulta. La consulta puede contener una subconsulta, una combinación de tablas o cualquier sentencia SELECT válida. Las restricciones CON NOMBRE no se crean en una tabla desde la otra, sólo se crean aquellas restricciones que carecen de nombre.



Caso práctico

12 Para crear la tabla EJEMPLO_AS a partir de los datos de la tabla EJEMPLO se procede así:

```
CREATE TABLE EJEMPLO_AS
AS SELECT * FROM EJEMPLO;
```

La tabla se crea con los mismos nombres de columnas e idéntico contenido de filas que la tabla EJEMPLO.

En el siguiente ejemplo, asignamos un nombre a las columnas de la tabla EJEMPLO_AS2 (COL1, COL2, COL3 y COL4):

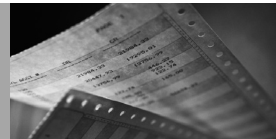
```
CREATE TABLE EJEMPLO_AS2 (COL1, COL2, COL3, COL4)
AS SELECT * FROM EJEMPLO;
```

El contenido de la tabla es el mismo que el que tiene la tabla EJEMPLO.

Se crea la tabla EMPLEYDEPART a partir de las tablas EMPLE y DEPART. Esta tabla contendrá el apellido y el nombre del departamento de cada empleado:

```
CREATE TABLE EMPLEYDEPART
AS SELECT APELLIDO, DNOMBRE
FROM EMPLE, DEPART WHERE EMPLE.DEPT_NO = DEPART.DEPT_NO;
```

A continuación, se listan las restricciones para las tablas EJEMPLO y EJEMPLO_AS: `SELECT CONSTRAINT_NAME, TABLE_NAME FROM USER_CONSTRAINTS WHERE TABLE_NAME IN ('EJEMPLO', 'EJEMPLO_AS');` Observamos que en la tabla EJEMPLO_AS sólo aparecen dos restricciones sin nombre, correspondientes a la restricción NOT NULL y definidas para las dos primeras columnas de la tabla, que son el DNI y el NOMBRE.




7.3 Supresión de tablas

La orden SQL DROP TABLE suprime una tabla de la base de datos. Cada usuario puede borrar sus propias tablas; sólo el administrador de la base de datos o algún usuario con el privilegio DROP ANY TABLE puede borrar las tablas de otro usuario. Al suprimir una tabla también se suprimen los índices y los privilegios asociados a ella. Las vistas y los sinónimos creados a partir de esta tabla dejan de funcionar, pero siguen existiendo en la base de datos, por lo que habría que eliminarlos. El formato de la orden DROP TABLE es:

```
DROP TABLE [usuario].nombretabla [CASCADE CONSTRAINTS];
```

CASCADE CONSTRAINTS elimina las restricciones de integridad referencial que remitan a la clave primaria de la tabla borrada.

Caso práctico



13 Recordemos ahora la tabla PROVINCIAS, que tiene definida clave primaria en la columna CODPROVINCIA, y la tabla PERSONAS, que tiene definida una clave ajena (CODPROVIN) referenciando a la tabla PROVINCIAS.

Si intentamos borrar la tabla PROVINCIAS, Oracle nos dará un mensaje de error:

```
DROP TABLE PROVINCIAS;
          *
ERROR en línea 1:
ORA-02449: claves únicas/primarias en la tabla referidas por claves ajenas
```

El error se debe a que existe una restricción de clave ajena en la tabla PERSONAS que referencia a la clave primaria de la tabla PROVINCIAS. Para borrar esta tabla hay que usar la opción CASCADE CONSTRAINTS, que suprime todas las restricciones de integridad referencial que se refieran a claves de la tabla borrada:

```
DROP TABLE PROVINCIAS CASCADE CONSTRAINTS;
```

Orden TRUNCATE

Permite suprimir todas las filas de una tabla y liberar el espacio ocupado para otros usos sin que desaparezca la definición de la tabla de la base de datos. Es una orden del lenguaje de definición de datos o DDL y no genera información de retroceso (ROLLBACK); es decir, una sentencia TRUNCATE no se puede anular, como tampoco activa disparadores DELETE. Por eso, la eliminación de filas con la orden TRUNCATE es más rápida que con DELETE. Su formato es:

```
TRUNCATE TABLE [usuario.]nombretabla [{DROP|REUSE} STORAGE];
```

La siguiente sentencia borra todas las filas de la tabla EJEMPLO:

```
TRUNCATE TABLE EJEMPLO;
```

De forma opcional, TRUNCATE permite liberar el espacio utilizado por las filas suprimidas. Con la opción DROP STORAGE se libera todo el espacio, excepto el especificado mediante el parámetro MINEXTENTS de la tabla; se trata de la opción por defecto. REUSE STORAGE mantendrá reservado el espacio para nuevas filas de la tabla.

No se puede truncar una tabla cuya clave primaria sea referenciada por la clave ajena de otra tabla. Antes de truncar la tabla hay que desactivar la restricción. Ejemplo:

```
TRUNCATE TABLE PROVINCIAS;
ERROR en línea 1:
ORA-02266: claves únicas/primarias en la tabla referi-
das por claves ajenas activadas
```

7.4 Modificación de tablas

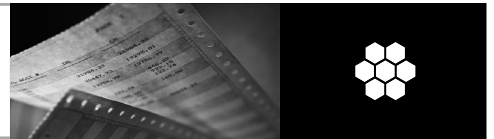
Se puede modificar una tabla mediante la orden **ALTER TABLE.** La modificación de tablas nos permitirá: añadir, modificar o eliminar columnas de una tabla existente, añadir o eliminar restricciones y activar o desactivar restricciones.

```
ALTER TABLE nombretabla
{[ADD (columna)
[MODIFY (columna [,columna] ...) ]
[DROP COLUMN (columna [,columna] ...) ]
[ADD  CONSTRAINT restricción]
[DROP CONSTRAINT restricción]
[DISABLE CONSTRAINT restricción]
[ENABLE CONSTRAINT restricción]};
```

A. Añadir, modificar o eliminar columnas a una tabla

Veamos a continuación cómo se usa la orden ALTER TABLE para añadir, modificar o eliminar columnas de una tabla.

- **Add:** Se utiliza ADD para añadir columnas a una tabla. A la hora de añadir una columna a una tabla hay que tener en cuenta varios factores:
  - Si la columna no está definida como NOT NULL, se le puede añadir en cualquier momento.
  - Si la columna está definida como NOT NULL, cabe la posibilidad de seguir los siguientes pasos: en primer lugar se añade la columna sin especificar NOT NULL; después se da valor a la columna para cada una de las filas; finalmente, se modifica la columna a NOT NULL.
- **Modify:** Modifica una o más columnas existentes en la tabla. Al modificar una columna de una tabla se han de tener en cuenta estos aspectos:
  - Se puede aumentar la longitud de una columna en cualquier momento.



- Al disminuir la longitud de una columna que tiene datos no se puede dar menor tamaño que el máximo valor almacenado.
- Es posible aumentar o disminuir el número de posiciones decimales en una columna de tipo NUMBER.
- Si la columna es NULL en todas las filas de la tabla, se puede disminuir la longitud y modificar el tipo de dato.
- La opción MODIFY ... NOT NULL sólo será posible cuando la tabla no contenga ninguna fila con valor nulo en la columna que se modifica.

• **Drop column:** Se utiliza para borrar una columna de una tabla. Hay que tener en cuenta que no se pueden borrar todas las columnas de una tabla y tampoco se pueden eliminar claves primarias referenciadas por claves ajenas.

B. Adición y borrado de restricciones

Podemos añadir y eliminar las siguientes restricciones de una tabla: CHECK, PRIMARY KEY, NOT NULL, FOREIGN KEY y UNIQUE.

Para añadir restricciones usamos la orden: `ALTER TABLE` *`nombretabla`* **`ADD CONSTRAINT`** *`nombrerestricción`* `...`

Para eliminar restricciones usamos la orden: `ALTER TABLE` *`nombretabla`* **`DROP CONSTRAINT`** *`nombrerestriccion`* `...`

Se pueden eliminar las restricciones con nombre y las asignadas por el sistema (SYS_C00n).

Casos prácticos



14 **Supongamos que la tabla EJEMPLO3 tiene datos y queremos añadir dos columnas: SEXO con la restricción NOT NULL e IMPORTE. Ocurrirá un error ya que la tabla no está vacía:**

```
ALTER TABLE EJEMPLO3 ADD(SEXO CHAR(1) NOT NULL, IMPORTE NUMBER(4));
ERROR en línea 1:
ORA-01758: la tabla debe estar vacía para agregar la columna (NOT NULL)
obligatoria
```

La opción ADD ... NOT NULL sólo será posible si la tabla está vacía. La solución a este problema consiste en modificar la tabla añadiendo la columna sin restricción:

```
ALTER TABLE EJEMPLO3 ADD (SEXO CHAR(1), IMPORTE NUMBER(4));
```

A continuación, modifica la columna dándole un valor, sea verdadero o no:

```
UPDATE EJEMPLO3 SET SEXO = 'X';
```

(Continúa)

(Continuación)

Por último, se vuelve a modificar la tabla con la opción MODIFY y se cambia la definición de la columna a NOT NULL.

Eliminamos las columnas SEXO e IMPORTE de la tabla EJEMPLO3, primero se elimina una columna:

ALTER TABLE EJEMPLO3 **DROP COLUMN** SEXO; y luego la otra: **ALTER TABLE** EJEMPLO3 **DROP COLUMN** IMPORTE;

15 **Añadiendo una restricción CHECK. Añadimos una restricción a la tabla EMPLE indicando que el SALARIO ha de ser > 0:**

```
ALTER TABLE EMPLE ADD CONSTRAINT SALMAYOR CHECK (SALARIO > 0);
```

Añadiendo una restricción UNIQUE. Añadimos la restricción de APELLIDO único a la tabla EMPLE: ALTER TABLE EMPLE **ADD CONSTRAINT** APELLIDO_UQ **UNIQUE**(APELLIDO);

Añadiendo una restricción NOT NULL. Añadimos la restricción de COMISION no nula a la tabla EMPLE:

```
ALTER TABLE EMPLE MODIFY COMISION CONSTRAINT COMI_NONULA NOT NULL;
ERROR en línea 1:
ORA-02296: no se puede activar (SCOTT.COMI_NONULA) - se han encontrado valores Nulos
```

En este ejemplo, aparece un error debido a que la columna COMISION es nula en muchas filas de la tabla; para añadir la restricción es necesario dar valores a COMISION para todas las filas de la tabla.

Añadiendo una restricción PRIMARY KEY. Añadimos la restricción de clave primaria a la columna EMP_NO de la tabla EMPLE: ALTER TABLE EMPLE **ADD CONSTRAINT** PK_EMPLE **PRIMARY KEY**(EMP_NO); Añadimos la restricción de clave primaria a la columna DEPT_NO de la tabla DEPART: ALTER TABLE DEPART **ADD CONSTRAINT** PK_DEPART **PRIMARY KEY**(DEPT_NO);

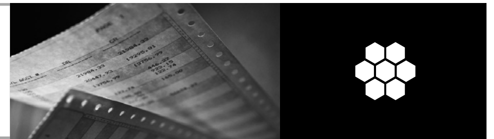
Añadiendo una restricción FOREIGN KEY. Añadimos la restricción de clave ajena a la columna DEPT_NO de la tabla EMPLE que referencia a la clave primaria de la tabla DEPART: ALTER TABLE EMPLE **ADD CONSTRAINT** FK_EMPLE **FOREIGN KEY**(DEPT_NO) **REFERENCES** DEPART ON DELETE CASCADE;

Veamos ahora las restricciones definidas para cada columna de la tabla EMPLE: SELECT CONSTRAINT_NAME, COLUMN_NAME FROM USER_CONS_COLUMNS WHERE TABLE_NAME = 'EMPLE';

```
CONSTRAINT_NAME          COLUMN_NAME
---------------        ---------------
 SYS_C005311              EMP_NO
 SYS_C005312              DEPT_NO
 APELLIDO_UQ              APELLIDO
 SALMAYOR                 SALARIO
 PK_EMPLE                 EMP_NO
 FK_EMPLE                 DEPT_NO
```

Eliminamos algunas de las restricciones definidas en la tabla EMPLE:

```
ALTER TABLE EMPLE DROP CONSTRAINT SYS_C005311;
ALTER TABLE EMPLE DROP CONSTRAINT APELLIDO_UQ,
```



C. Activar y desactivar restricciones

Por defecto, las restricciones se activan al crearlas. Se pueden desactivar añadiendo la cláusula DISABLE al final de la restricción. El siguiente ejemplo añade una restricción, pero la desactiva: `ALTER TABLE EMPLE ADD CONSTRAINT APELLIDO_UQ UNIQUE(APELLIDO) DISABLE;`

Para desactivar una restricción existente usamos la orden:

```
ALTER TABLE nombretabla  DISABLE CONSTRAINT nombreres-
tricción ...
```

Para activar una restricción existente usamos la orden:

```
ALTER TABLE nombretabla  ENABLE CONSTRAINT nombreres-
tricción ...
```

Caso práctico



16 **Desactivamos algunas restricciones de la tabla EMPLE:**

```
ALTER TABLE EMPLE DISABLE CONSTRAINT PK_EMPLE;
ALTER TABLE EMPLE DISABLE CONSTRAINT FK_EMPLE,
```

Actividades propuestas



7 **Crea la tabla TIENDAS sin restricciones; la descripción es la siguiente:**

```
NIF          VARCHAR2(10)      POBLACION     VARCHAR2(20)
NOMBRE       VARCHAR2(20)      PROVINCIA     VARCHAR2(20)
DIRECCION    VARCHAR2(20)      CODPOSTAL     NUMBER(5)
```

Después añade la siguientes restricciones:

- La clave primaria es NIF.
- PROVINCIA ha de almacenarse en mayúscula.
- Cambia la longitud de NOMBRE a 30 caracteres y no nulo.

7.5 Creación y uso de vistas

A veces, para obtener datos de varias tablas hemos de construir una sentencia SELECT compleja y, si en otro momento necesitamos realizar esa misma consulta, tenemos que construir de nuevo la sentencia SELECT. Sería muy cómodo obtener los datos de una consulta compleja con una simple sentencia SELECT.

Pues bien, las vistas solucionan este problema: mediante una consulta simple de una vista cabe la posibilidad de obtener datos de una consulta compleja. Una **vista** es una tabla lógica que permite acceder a la información de una o de varias tablas. No contiene información por sí misma, sino que su información está basada en la que contienen otras tablas, llamadas *tablas base*, y siempre refleja los datos de estas tablas; es, simplemente, una sentencia SQL.

Si se suprime una tabla, la vista asociada se invalida. Las vistas tienen la misma estructura que una tabla: filas y columnas, y se tratan de forma semejante a una tabla. El formato para crear una vista es:

```
CREATE [OR REPLACE] VIEW Nombrevista [(columna [,columna])]
AS consulta
[WITH {CHECK OPTION | READ ONLY} CONSTRAINT nombrerestric-
                                            ción];
```

Nombrevista es el nombre que damos a la vista.

[(*columna* [,*columna*])] son los nombres de columnas que va a contener la vista. Si no se ponen, se asumen los nombres de columna devueltos por la consulta.

AS *consulta* determina las columnas y las tablas que aparecerán en la vista.

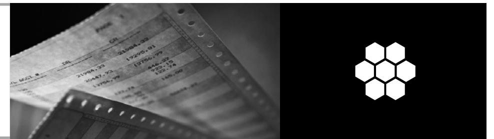
[OR REPLACE] crea de nuevo la vista si ya existía.

[**WITH CHECK OPTION**] es la opción de comprobación para una vista. Si se especifica, SQL comprueba automáticamente cada operación INSERT y UPDATE sobre la vista para asegurarse que las filas resultantes satisfagan el criterio de búsqueda de la definición de la vista.

Si la fila insertada o modificada no satisface la condición de creación de la vista, la sentencia INSERT o UPDATE falla y no se realiza la operación.

[**WITH READ ONLY**] especifica que sólo se puede hacer SELECT de las filas de la vista.

[**CONSTRAINT** *nombrerestriccion*] especifica el nombre de la restricción WITH CHECK OPTION o WITH READ ONLY. Es opcional.



A. Creación y uso de vistas sencillas

Las vistas más sencillas son las que acceden a una única tabla. Por ejemplo, creamos la vista DEP30 que contiene el APELLIDO, el OFICIO y el SALARIO de los empleados de la tabla EMPLE del departamento 30:

```
CREATE VIEW DEP30
AS SELECT APELLIDO, OFICIO, SALARIO  FROM EMPLE
   WHERE DEPT_NO=30;
```

Ahora la vista creada se puede usar como si de una tabla se tratase. Se puede consultar, se pueden borrar filas, actualizar filas siempre y cuando las columnas a actualizar no sean expresiones (funciones de grupo o referencias a pseudocolumnas); y se puede insertar siempre y cuando todas las columnas obligatorias de la tabla asociada estén presentes en la vista.

También podríamos haberla creado dando nombre a las columnas, por ejemplo, APE, OFI y SAL:

```
CREATE OR REPLACE VIEW DEP30 (APE, OFI, SAL)
AS SELECT APELLIDO, OFICIO, SALARIO FROM EMPLE
   WHERE DEPT_NO = 30;
```

Para consultar las vistas creadas se dispone de la vista **USER_VIEWS** y **ALL_VIEWS**. Así, para visualizar los nombres de vistas con sus textos, tenemos:

```
SQL> SELECT VIEW_NAME, TEXT FROM USER_VIEWS;
```

Actividades propuestas



8 **Crea una vista que contenga los datos de los empleados del departamento 10 con salario > 1200. Después realiza operaciones INSERT, UPDATE y DELETE sobre la vista.**

B. Creación y uso de vistas con WITH CHECK OPTION y READ ONLY

Se puede crear una vista de forma que todas las operaciones INSERT, UPDATE y DELETE que se hagan sobre ella satisfagan la condición por la que se creó. Por ejemplo, creo una vista que contiene todos los datos de los empleados del departamento 20:

```
CREATE OR REPLACE  VIEW DEP20
AS SELECT * FROM EMPLE WHERE DEPT_NO = 20;
```

Ahora inserto en la vista un empleado del departamento 30:

```
INSERT INTO DEP20 VALUES (3333, 'PEREZ', 'EMPLEADO',7902,
SYSDATE, 1300, NULL, 30);
```

La inserción se realizará correctamente.

Ahora creamos la vista con WITH CHECK OPTION:

```
CREATE OR REPLACE VIEW DEP20
AS SELECT * FROM EMPLE WHERE DEPT_NO = 20 WITH CHECK
OPTION;
```

Intentamos insertar una fila en la vista con el departamento 30; en este caso se producirá un error:

```
ERROR en línea 1:
ORA-01402: violación de la cláusula WHERE en la vista
WITH CHECK OPTION
```

La opción **WITH READ ONLY** sólo nos permitirá hacer SELECT en la vista:

```
CREATE OR REPLACE VIEW DEP30
AS SELECT * FROM EMPLE WHERE DEPT_NO=30 WITH READ ONLY;
```

Cualquier operación INSERT, UPDATE o DELETE sobre la vista DEP30 fallará.

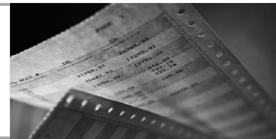


Actividades propuestas

9 **Prueba a realizar operaciones INSERT, UPDATE o DELETE sobre la vista DEP30.**

C. Creación y uso de vistas complejas

Las vistas complejas contienen consultas que se definen sobre más de una tabla, agrupan filas usando las cláusulas GROUP BY o DISTINCT, y contienen llamadas a funciones. Se pueden crear vistas usando funciones, expresiones en columnas y consultas avanzadas, pero únicamente se podrán consultar estas vistas.




Caso práctico



17 **A partir de las tablas EMPLE y DEPART creamos una vista que contenga las columnas EMP_NO, APELLIDO, DEPT_NO y DNOMBRE :**

```
CREATE VIEW EMP_DEPT (EMP_NO, APELLIDO, DEPT_NO, DNOMBRE) AS
   SELECT EMP_NO,APELLIDO, EMPLE.DEPT_NO, DNOMBRE
   FROM EMPLE, DEPART WHERE EMPLE.DEPT_NO = DEPART.DEPT_NO;
```

Insertamos una fila en la vista: `INSERT INTO EMP_DEPT VALUES(2222,'SUELA', 20, 'INVESTIGACION');` Pero se produce un error debido a que la vista se creó a partir de dos tablas.

```
ERROR en línea 1:
ORA-01776: no se puede modificar más de una tabla base a través de una vista de unión
```

Los borrados y las modificaciones también producirán errores.

Creamos una vista llamada PAGOS a partir de las filas de la tabla EMPLE, cuyo departamento sea el 10. Las columnas de la vista se llamarán NOMBRE, SAL_MES, SAL_AN y DEPT_NO. El NOMBRE es la columna APELLIDO, a la que aplicamos la función INITCAP( ). SAL_MES es el SALARIO. SAL_AN es el SALARIO*12:

```
CREATE VIEW PAGOS (NOMBRE, SAL_MES, SAL_AN, DEPT_NO)
AS SELECT INITCAP(APELLIDO), SALARIO, SALARIO*12, DEPT_NO FROM EMPLE WHERE DEPT_NO = 10;
```

Podemos modificar filas siempre y cuando la columna que se va a modificar no sea la columna expresada en forma de cálculo (SAL_AN) o la que fue creada mediante la función INITCAP(): `UPDATE PAGOS SET SAL_MES = 5000 WHERE NOMBRE = 'Muñoz';`

Actividades propuestas



10 **Crea la vista VMEDIA a partir de las tablas EMPLE y DEPART. Esta vista contendrá por cada departamento el número de departamento, el nombre, la media de salario y el máximo salario. Prueba hacer inserciones modificaciones y borrados en la vista.**

D. Borrado de vistas

Es posible borrar las vistas con la orden DROP VIEW, cuyo formato es:

```
DROP VIEW nombrevista;
```

Por ejemplo, borramos la vista PAGOS: `DROP VIEW PAGOS;`

7.6 Creación de sinónimos

Cuando tenemos acceso a las tablas de otro usuario y deseamos consultarlas es preciso teclear el nombre del usuario propietario antes del nombre de la tabla. Es decir, si DIEGO tiene acceso a la tabla DEPART de PEDRO y la quiere consultar, tendrá que teclear la siguiente orden para poder hacerlo: `SELECT * FROM` **`PEDRO.DEPART;`**

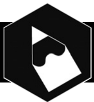
Mediante el uso de sinónimos, DIEGO puede crear un sinónimo para referirse a la tabla de PEDRO sin necesidad de incluir su nombre: `SELECT * FROM` **`TABLAPEDRO;`**

Un **sinónimo** es un nuevo nombre que se puede dar a una tabla o vista. Con los sinónimos se pueden utilizar dos nombres diferentes para referirse a un mismo objeto. Resultan interesantes cuando se tiene acceso a tablas de otros usuarios; se pueden crear sinónimos para hacer referencia a esas tablas y, así, no hay que escribir el nombre de usuario propietario de la tabla delante de la tabla a la que tenemos acceso cada vez que deseemos consultarla.

El formato para crear sinónimos es el siguiente:

CREATE [PUBLIC] **SYNONYM** *nombresinónimo* **FOR** [*usuario*.]*Nombretabla*;

`PUBLIC` hace que el sinónimo esté disponible para todos los usuarios.



Caso práctico

18 **Creamos el sinónimo DEPARTAMENTOS asociado a la tabla DEPART:** **CREATE SYNONYM** DEPARTAMENTOS **FOR** DEPART**;** Ahora podemos acceder a la tabla DEPART mediante su nombre o usando el sinónimo:

```
SELECT * FROM DEPARTAMENTOS;        SELECT * FROM DEPART;
```

DIEGO puede acceder a la tabla DEPART de PEDRO y crea un sinónimo llamado DEPART:

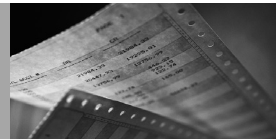
```
CREATE SYNONYM DEPART FOR PEDRO.DEPART;
```

Diego puede utilizar el sinónimo DEPART para consultar la tabla DEPART de PEDRO: `SELECT * FROM DEPART;`

Como DIEGO ha nombrado al sinónimo igual que el nombre que tiene la tabla de PEDRO (DEPART), podrá hacer uso de las aplicaciones que PEDRO ha desarrollado sobre esa tabla.

Por otra parte, existen sinónimos públicos a los que puede hacer referencia cualquier usuario. Sólo el administrador de la base de datos (DBA) y los usuarios con privilegio CREATE PUBLIC SYNONYM pueden crear este tipo de sinónimos. Por ejemplo, un usuario que es DBA crea un sinónimo público para su tabla DEPART; llama DEP al sinónimo:

```
CREATE PUBLIC SYNONYM DEP FOR DEPART;
```





Para que todos los usuarios puedan usar la tabla DEPART y su sinónimo han de tener permiso. Se da permiso a todos los usuarios para hacer SELECT en la tabla DEPART con la orden GRANT: **GRANT SELECT** ON DEPART TO **PUBLIC**; ahora todos los usuarios pueden hacer SELECT del sinónimo público creado para la tabla DEPART: SELECT * FROM **DEP**;

A. Borrado de sinónimos

Del mismo modo que se crean sinónimos, se pueden borrar, con la orden DROP SYNONYM, cuyo formato es:

```
DROP [PUBLIC] SYNONYM [usuario.]sinónimo;
```

sinónimo es el nombre de sinónimo que se va a suprimir. Únicamente los DBA y los usuarios con el privilegio DROP PUBLIC SYNONYM pueden suprimir sinónimos PUBLIC. Igualmente, sólo los DBA y los usuarios con el privilegio DROP ANY SYNONYM pueden borrar los sinónimos de otros usuarios. Por ejemplo, borramos el sinónimo DEPARTAMENTOS:

```
DROP SYNONYM DEPARTAMENTOS;
```

Borramos el sinónimo público DEP: **DROP PUBLIC SYNONYM** DEP;

Por otro lado, la vista USER_SYNONYMS permite ver los sinónimos que son propiedad del usuario. Para ver los sinónimos creados por el usuario sobre sus objetos: SELECT SYNONYM_NAME, TABLE_NAME FROM **USER_SYNONYMS;**

7.7 Cambios de nombre

RENAME es una orden SQL que cambia el nombre de una tabla, vista o sinónimo. El nuevo nombre no puede ser una palabra reservada ni el nombre de un objeto que tenga creado el usuario. El formato es éste:

```
RENAME nombreanterior TO nombrenuevo;
```

Las restricciones de integridad, los índices y los permisos dados al objeto se transfieren automáticamente al nuevo objeto. Oracle invalida todos los objetos que dependen del objeto renombrado, como las vistas, los sinónimos y los procedimientos almacenados que hacen referencia a la tabla renombrada. No se puede usar esta orden para renombrar sinónimos públicos ni para renombrar columnas de una tabla. Las columnas de una tabla se renombran mediante la orden CREATE TABLE AS.

Conceptos básicos



A continuación se muestra un resumen sobre la orden CREATE TABLE. El formato más básico es el siguiente:

```
CREATE  TABLE  Nombretabla
(
  Columna1   Tipo_dato  [NOT NULL],
  Columna2   Tipo_dato  [NOT NULL],
  ...
) [TABLESPACE espacio_de_tabla];
```

Restricciones en la orden CREATE TABLE:

- Restricción de un solo campo:

```
CONSTRAINT nombrerestricción {
    [NOT] NULL | {PRIMARY KEY | UNIQUE} |
    REFERENCES Nombretabla [(columna[, columna…])]  [ON DELETE CASCADE] |
    CHECK (condición)
}
```

- Restricción de múltiples campos:

```
CONSTRAINT nombrerestricción {
     PRIMARY KEY (columna[, columna …]) |
     UNIQUE (columna[, columna …]) |
     FOREIGN KEY (columna[, columna …])
     REFERENCES Nombretabla [(columna[, columna …])] [ON DELETE CASCADE] |
     CHECK (condición)
}
```

VISTAS DEL DICCIONARIO DE DATOS:

Información de tablas y otros objetos	USER_TABLES, USER_OBJECTS, USER_CATALOG
Información de restricciones	USER_CONSTRAINTS, ALL_CONSTRAINTS, DBA_CONSTRAINTS USER_CONS_COLUMNS, ALL_CONS_COLUMNS, DBA_CONS_COLUMNS
Información sobre vistas	USER_VIEWS, ALL_VIEWS
Información sobre sinónimos	USER_SYNONYMS, ALL_SYNONYMS



Actividades complementarias

1 **Crea las siguientes tablas de acuerdo con las restricciones que se mencionan:**

Tabla PEDIDOS

Descripción de la tabla:

```
NIF                 VARCHAR2(10)
ARTICULO            VARCHAR2(20)
COD_FABRICANTE      NUMBER(3)
PESO                NUMBER(3)
CATEGORIA           VARCHAR2(10)
FECHA_PEDIDO        DATE
UNIDADES_PEDIDAS    NUMBER(4)
```

- La clave primaria está formada por las columnas: NIF, ARTICULO, COD_FABRICANTE, PESO, CATEGORIA y FECHA_PEDIDO.
- COD_FABRICANTE es clave ajena que referencia a la tabla FABRICANTES.
- UNIDADES_PEDIDAS ha de ser > 0.
- CATEGORIA ha de ser 'Primera', 'Segunda' o 'Tercera'.
- Las columnas ARTICULO, COD_FABRICANTE, PESO y CATEGORIA son clave ajena y referencian a la tabla ARTICULOS. Realiza un borrado en cascada.
- NIF es clave ajena y referencia a la tabla TIENDAS.

Tabla VENTAS

Descripción de la tabla:

```
NIF                 VARCHAR2(10)
ARTICULO            VARCHAR2(20)
COD_FABRICANTE      NUMBER(3)
PESO                NUMBER(3)
CATEGORIA           VARCHAR2(10)
FECHA_VENTA         DATE
UNIDADES_VENDIDAS   NUMBER(4)
```

- La clave primaria está formada por las columnas: NIF, ARTICULO, COD_FABRICANTE, PESO, CATEGORIA y FECHA_VENTA.
- COD_FABRICANTE es clave ajena que referencia a la tabla FABRICANTES.
- UNIDADES_VENDIDAS ha de ser > 0.
- CATEGORIA ha de ser 'Primera', 'Segunda' o 'Tercera'.
- Las columnas ARTICULO, COD_FABRICANTE, PESO y CATEGORIA son clave ajena y referencian a la tabla ARTICULOS. Realizar un borrado en cascada.
- NIF es clave ajena y referencia a la tabla TIENDAS.

2 **Visualiza las restricciones definidas para las tablas anteriores.**

3 **Modifica las columnas de las tablas PEDIDOS y VENTAS para que las UNIDADES_VENDIDAS y las UNIDADES_PEDIDAS puedan almacenar cantidades numéricas de 6 dígitos.**

4 **A partir de la tabla TIENDAS impide que se den de alta más tiendas en la provincia de 'TOLEDO'.**

5 **Añade a las tablas PEDIDOS y VENTAS una nueva columna para que almacenen el PVP del artículo.**

Tablas PERSONAL, PROFESORES Y CENTROS

6 **Crea una vista que se llame CONSERJES que contenga el nombre del centro y el nombre de sus conserjes.**

7 **Crea un sinónimo llamado CONSER asociado a la vista creada antes.**

8 **Añade a la tabla PROFESORES una columna llamada COD_ASIG con dos posiciones numéricas.**

9 **Crea la tabla TASIG con las siguientes columnas: COD_ASIG numérico, 2 posiciones y NOM_ASIG cadena de 20 caracteres.**

10 **Añade la restricción de clave primaria a la columna COD_ASIG de la tabla TASIG.**

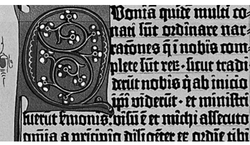
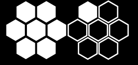
11 **Añade la restricción de clave ajena a la columna COD_ASIG de la tabla PROFESORES. Visualiza el nombre de las restricciones y las columnas y las columnas afectadas para las tablas TASIG y PROFESORES.**

Introducción al lenguaje PL/SQL

8

En esta unidad aprenderás a:

1. **Identificar los distintos tipos de programas y los objetos que maneja PL/SQL.**
2. **Reconocer las principales características del lenguaje, sus posibilidades de utilización y sus limitaciones.**
3. **Manejar la estructura básica del lenguaje, el bloquey reconocer sus componentes.**
4. **Comprender el funcionamiento de programas sencillos.**
5. **Utilizar las posibilidades del entorno de SQL*Plus para la edición, depuración y ejecución de programas sencillos en PL/SQL.**

8.1 Introducción

Hasta el momento, hemos trabajado con la base de datos de manera interactiva. Esta forma de trabajar, incluso con un lenguaje tan sencillo y potente como SQL, no resulta operativa en un entorno de producción, ya que supondría que todos los usuarios conocen y manejan SQL, y además está sujeta a frecuentes errores.

También hemos creado pequeños *scripts* de instrucciones SQL y SQL*Plus. Pero estos *scripts* tienen importantes limitaciones en cuanto al control de la secuencia de ejecución de instrucciones, el uso de variables, la modularidad, la gestión de posibles errores, etcétera.

Para superar estas limitaciones, Oracle incorpora un gestor PL/SQL en el servidor de la base de datos. Este lenguaje, basado en el lenguaje ADA, incorpora todas las características propias de los **lenguajes de tercera generación:** manejo de variables, estructura modular (procedimientos y funciones), estructuras de control (bifurcaciones, bucles y demás estructuras), control de excepciones. También incorpora un completo soporte para la **Programación Orientada a Objetos (POO),** por lo que puede ser considerado como un lenguaje procedimental y orientado a objetos.

Los programas creados con PL/SQL se pueden almacenar en la base de datos como cualquier otro objeto quedando así disponibles para su ejecución por los usuarios. Esta forma de trabajo facilita la instalación, distribución y mantenimiento de los programas y reduce los costes asociados a estas tareas. Además, la ejecución de los programas en el servidor, conlleva un ahorro de recursos en los clientes y disminución del tráfico de red.

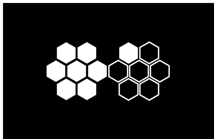
El uso del lenguaje PL/SQL es también imprescindible para construir disparadores de bases de datos que permiten implementar reglas complejas de negocio y auditoría en la base de datos.

8.2 Características del lenguaje

Por todo ello, el conocimiento de este lenguaje es imprescindible para poder trabajar en el entorno Oracle, tanto para administradores de la base de datos como para desarrolladores de aplicaciones. En las próximas unidades estudiaremos los fundamentos del lenguaje y su aplicación al servidor de la base de datos.

PL/SQL es un lenguaje procedimental diseñado por Oracle para trabajar con la base de datos. Está incluido en el servidor y en algunas herramientas de cliente. Soporta todos los comandos de consulta y manipulación de datos.

La unidad de trabajo es el **bloque**, constituido por un conjunto de declaraciones, instrucciones y mecanismos de gestión de errores y excepciones.



A. Bloques PL/SQL

El **bloque** es la estructura básica característica de todos los programas PL/SQL. Tiene tres zonas claramente definidas:

- Una **zona de declaraciones** donde se declaran objetos locales (variables, constantes, etcétera). Suele ir precedida por la cláusula DECLARE (o IS/AS en los procedimientos y funciones). Es opcional.
- Un **conjunto de instrucciones** precedido por la cláusula BEGIN.
- Una **zona de tratamiento de excepciones** precedido por la cláusula EXCEPTION. Esta zona, igual que la de declaraciones, es opcional.

El formato genérico del bloque es:

```
[ DECLARE
    <declaraciones> ]
BEGIN
    <órdenes>
[ EXCEPTION
    <gestión de excepciones> ]
END;
```

La zona de declaraciones comenzará con DECLARE o con IS, dependiendo del tipo de bloque. Las únicas cláusulas obligatorias son BEGIN y END.

Caso práctico

1 **En el siguiente ejemplo se borra el departamento número 20, pero evitando posibles errores por violar restricciones de integridad referencial, pues el departamento tiene empleados asociados. Para ello crearemos antes un departamento provisional, al que asignamos los empleados del departamento 20 antes de borrar dicho departamento. El programa también informa del número de empleados afectados.**

Antes de introducir el ejemplo deberemos introducir la instrucción SET SERVEROUTPUT ON o utilizar el menú de configuración de SQL*Plus para que muestre los mensajes.

```
DECLARE
    v_num_empleados NUMBER(2);
BEGIN
    INSERT INTO depart VALUES (99,'PROVISIONAL',NULL);
    UPDATE emple SET dept_no = 99
      WHERE dept_no = 20;
    v_num_empleados := SQL%ROWCOUNT;
    DELETE FROM depart
      WHERE dept_no = 20;
     DBMS_OUTPUT.PUT_LINE(v_num_empleados ||
         '  Empleados ubicados en PROVISIONAL');
```

(Continúa)



(Continuación)

```
EXCEPTION
    WHEN OTHERS THEN
      RAISE_APPLICATION_ERROR(-20000, 'Error en aplicación');
END;
/
```

El resultado de la ejecución del programa será:

```
5  Empleados ubicados en PROVISIONAL

Procedimiento PL/SQL terminado con éxito.
```

La utilización de bloques supone una notable mejora de rendimiento ya que se envían los bloques completos al servidor para que sean procesados, en lugar de cada sentencia SQL. Así se ahorran muchas operaciones de E/S.

Los bloques PL/SQL se pueden anidar para conseguir distintas funcionalidades (por ejemplo, para evitar la propagación de excepciones) como veremos más adelante. A estos bloques se les llama **bloques anidados.**

```
DECLARE
    ...
BEGIN
    ...
    DECLARE -- comienzo del bloque interior anidado
        ...
    BEGIN
        ...
    EXCEPTION
        ...
    END;    -- fin del bloque interior anidado
    ...
EXCEPTION
    ...
END
```

Se pueden anidar bloques en las secciones ejecutable y de excepciones, pero no en la declarativa.

B. Definición de datos compatibles con SQL

PL/SQL dispone de tipos de datos compatibles con los tipos utilizados para las columnas de las tablas: NUMBER, VARCHAR2, DATE, etcétera, además de otros propios, como BOOLEAN.

Las declaraciones de los datos deben realizarse en la sección de declaraciones:

```
DECLARE
    importe  NUMBER(8,2);
```




```
      contador NUMBER(2);
      nombre   CHAR(20);
      nuevo    VARCHAR2(15);
   BEGIN
    ...
```

PL/SQL permite declarar una variable del mismo tipo que otra variable o que una columna de una tabla mediante el atributo %TYPE.

Ejemplo: `nombre_act empleados.nombre%TYPE;`

Declara la variable *nombre_act*, del mismo tipo que la columna nombre de la tabla *empleados*.

También se puede declarar una variable para guardar una fila completa de una tabla mediante el atributo %ROWTYPE.

Ejemplo: `mi_fila empleados%ROWTYPE;`

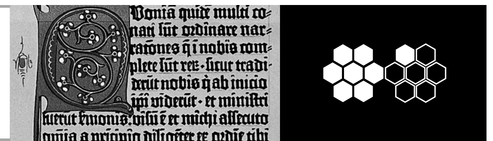
Declara la variable *mi_fila* del mismo tipo que las filas de la tabla *empleados*.

C. Estructuras de control

Las **estructuras de control** de PL/SQL son las habituales de los lenguajes de programación estructurados: IF, CASE, WHILE, FOR y LOOP.

Estructuras de control alternativas		
Alternativa simple IF <condición> THEN instrucciones; END IF; **Alternativa doble** IF <condición> THEN instrucciones; ELSE instrucciones; END IF;	**Alternativa múltiple (elsif)** IF <condición> THEN instrucciones; ELSIF <condición2> THEN instrucciones; ELSIF <condición3> THEN instrucciones; ... ELSE instrucciones; END IF;	**Alternativa múltiple (case)** CASE [<expresión>] WHEN <test1> THEN <instrucciones1>; WHEN <test2> THEN <instrucciones2>; WHEN <test3> THEN <instrucciones3>; [ELSE <otrasinstrucciones>;] END CASE;

Estructuras de control repetitivas		
Mientras WHILE <condición> LOOP Instrucciones; END LOOP;	**Para** FOR <variable> IN <mínimo>..<máximo> LOOP instrucciones; END LOOP;	**Iterar... fin iterar salir si...** LOOP instrucciones; EXIT WHEN <condición>; instrucciones; END LOOP;



D. Soporte para órdenes de manipulación de datos

Desde PL/SQL se puede ejecutar cualquier orden de manipulación de datos. Ejemplos:

```
DELETE FROM clientes WHERE nif = v_nif;
```

Borra de la tabla *clientes* la fila correspondiente al cliente cuyo NIF se especifica en la variable *v_nif.*

```
UPDATE productos SET stock_disponible := stock_disponi-
ble - unidades_vendidas WHERE producto_no = v_producto;
```

Actualiza en la tabla *productos* la columna *stock_disponible*, correspondiente al código de producto especificado en la variable *v_producto*, restándole el valor que se especifica en la variable *unidades_vendidas*.

```
INSERT INTO clientes VALUES(v_num, v_nom, v_loc, ...);
```

Añade a la tabla *clientes* una fila con los valores contenidos en las variables que se especifican.

E. Soporte para consultas

PL/SQL permite ejecutar cualquier consulta admitida por la base de datos. Pero cuando se ejecuta la consulta, el resultado no se muestra automáticamente en el terminal del usuario, sino que queda en un área de memoria denominada **cursor** a la que accederemos utilizando variables.

Por ejemplo, para obtener en PL/SQL el número total de empleados de la tabla del mismo nombre no podemos utilizar directamente la instrucción SQL correspondiente pues dará error:

```
SELECT COUNT(*) FROM emple;    -- ERROR
```

Utilizaremos una o más variables (declaradas previamente) junto con la cláusula INTO para poder acceder a los datos devueltos por nuestra consulta, tal como figura en el siguiente ejemplo: `SELECT COUNT(*)` **`INTO v_total_emple`** `FROM emple;`

Ejecuta la consulta y deposita el resultado en la variable `v_total_emple`, que suponemos declarada previamente. A este tipo de cursores se les denomina **cursores implícitos,** ya que no hay que declararlos. Es el tipo más sencillo, pero tiene ciertas limitaciones: la consulta deberá devolver una única fila, pues en caso contrario se producirá un error catalogado como TOO_MANY_ROWS.

El formato básico es:

```
SELECT <columna/s> INTO <variable/s> FROM <tabla> WHERE...];
```

La/s variable/s que siguen a INTO reciben el valor de la consulta. Por tanto, debe haber coincidencia en el tipo con las columnas especificadas en la cláusula SELECT.


Caso práctico

2 **El siguiente bloque visualiza el apellido y el oficio del empleado cuyo número es 7900.**

```
DECLARE
    v_ape VARCHAR2(10);
    v_oficio VARCHAR2(10);
BEGIN
    SELECT apellido, oficio INTO v_ape, v_oficio
      FROM EMPLE WHERE EMP_NO = 7900;
    DBMS_OUTPUT.PUT_LINE(v_ape||'*'||v_oficio);
END;
/
```

El resultado de la ejecución del programa será:

```
JIMENO*EMPLEADO
Procedimiento PL/SQL terminado con éxito.
```

F. Gestión de excepciones

Las **excepciones** sirven para tratar errores y mensajes de aviso. En Oracle están disponibles excepciones predefinidas correspondientes a algunos de los errores más frecuentes que se producen al trabajar con la base de datos, como por ejemplo:

- NO_DATA_FOUND. Una orden de tipo SELECT INTO no ha devuelto ningún valor.
- TOO_MANY_ROWS. Una orden SELECT INTO ha devuelto más de una fila.

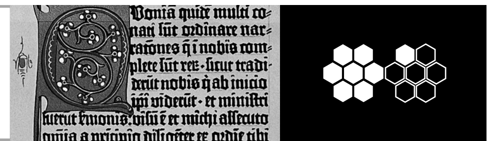
Las excepciones se disparan automáticamente al producirse los errores asociados. La sección EXCEPTION es la encargada de gestionar mediante los manejadores (WHEN) las excepciones que puedan producirse durante la ejecución del programa.


Caso práctico

3 **El ejemplo anterior, con gestión de excepciones, sería:**

```
DECLARE
    v_ape VARCHAR2(10);
    v_oficio VARCHAR2(10);
BEGIN
    SELECT apellido, oficio INTO v_ape, v_oficio
       FROM EMPLE WHERE EMP_NO = 7900;
       DBMS_OUTPUT.PUT_LINE(v_ape||'*'||v_oficio);
```

(Continúa)



(Continuación)

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20000,'ERROR no hay datos');
    WHEN TOO_MANY_ROWS THEN
        RAISE_APPLICATION_ERROR(-20000,'ERROR demasiados datos');
    WHEN OTHERS THEN
        RAISE_APPLICATION_ERROR(-20000,'Error en la aplicación');
END;
```

Cuando PL/SQL detecta una excepción, automáticamente pasa el control del programa a la sección EXCEPTION del bloque PL/SQL. Allí buscará un manejador (cláusula WHEN) para la excepción producida, o uno genérico (WHEN OTHERS), y realizará el tratamiento establecido. Al finalizar este tratamiento, sale del bloque actual y devuelve el control al programa o a la herramienta que realizó la llamada.

PL/SQL permite que el programador defina sus propias excepciones.

G. Estructura modular

En esta primera aproximación podemos distinguir los siguientes tipos de programas PL/SQL que se ejecutan el servidor Oracle:

- **Bloques anónimos**
  - No tienen nombre. Se ejecutan en el servidor pero no se guardan en él.
  - La zona de declaraciones comienza con la palabra DECLARE.
  - Son las estructuras utilizadas fundamentalmente en las primeras unidades de este libro por razones didácticas pero su utilización real es escasa.
  - Todos los ejemplos de programas vistos hasta el momento en esta unidad son bloques anónimos.
- **Subprogramas: procedimientos y funciones**
  - Son bloques PL/SQL que tienen un nombre por el que son invocados desde otros programas o herramientas.
  - Se compilan, almacenan y ejecutan en la base de datos Oracle.
  - Tienen una cabecera que incluye el nombre del subprograma (indicando si se trata de un procedimiento o una función), los parámetros y el tipo de valor de retorno en el caso de las funciones.
  - La zona de declaraciones y el bloque o cuerpo del programa comienza con la palabra IS o AS.

– Pueden ser de dos tipos: **PROCEDIMIENTOS** y **FUNCIONES.** Su formato genérico es prácticamente el mismo pero las funciones devuelven un valor en algún punto de su código y deben declarar el tipo de valor devuelto en la cabecera:

Procedimiento

```
PROCEDURE <nombreprocedimiento>
  [( <lista de parámetros> )]
IS
  [<declarac objetos locales>;]
BEGIN
   <instrucciones>;
[EXCEPTION
   <excepciones>;]
END [<nombreprocedimiento>];
```

Función

```
FUNCTION <nombrefunción>
   [( <lista de parámetros> )]
   RETURN <tipo valor devuelto >
IS
  [<declaración objetos locales>;]
BEGIN
   <instrucciones>;
    RETURN <expresión>;
[EXCEPTION
   <excepciones>;]
END [<nombrefuncion>];
```

Al final de esta unidad podemos ver ejemplos de procedimientos y funciones.

- **Disparadores de base de datos.**

  – Son programas almacenados en la base de datos que se asocian a un evento.

  – Se ejecutan automáticamente al producirse determinados cambios (inserción, borrado o modificación) en la tabla a la que están asociados o en el sistema.

  – Son muy útiles para controlar los cambios que suceden en la base de datos, para implementar restricciones complejas, etcétera.

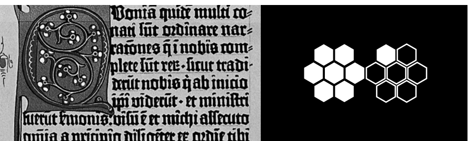
Hay un tipo de disparadores o *triggers* llamados **de sustitución** que se ejecutan en lugar de ciertas instrucciones de consulta sobre vistas.

Caso práctico

4 **El siguiente código crea un *trigger* que se ejecutará automáticamente cuando se elimine algún empleado en la tabla correspondiente visualizando el número y el nombre de los empleados borrados:**

```
CREATE OR REPLACE TRIGGER audit_borrado_emple
           BEFORE DELETE
           ON emple
       FOR EACH ROW
BEGIN
           DBMS_OUTPUT.PUT_LINE('BORRADO EMPLEADO'
                        || '*' ||:old.emp_no
                        ||'*'||:old.apellido);
END;
```

Estudiaremos con más detalle los disparadores o *triggers* de bases de datos en la unidad de programación avanzada en PL/SQL.



8.3 Interacción con el usuario en PL/SQL

PL/SQL es un lenguaje diseñado para trabajar con la base de datos y manejar grandes volúmenes de información de manera eficaz, pero no ha sido concebido para interactuar con el usuario. En efecto, PL/SQL no dispone de órdenes para la captura de datos introducidos por teclado, ni tampoco para visualizar datos en la pantalla. Para eso se utilizan otros lenguajes y herramientas. No obstante, Oracle incorpora el paquete DBMS_OUTPUT con fines de depuración. Éste incluye, entre otros, el procedimiento **PUT_LINE,** que permite visualizar textos en la pantalla.

Para aprender a programar necesitaremos probar frecuentemente los programas y visualizar sus resultados. Con este fin utilizaremos el procedimiento PUT_LINE, sabiendo que en un entorno de producción deberemos emplear otras herramientas especializadas para visualizar los resultados. El formato genérico para invocar a este procedimiento es el siguiente:

```
DBMS_OUTPUT.PUT_LINE(<expresión>);
```

Para que funcione correctamente, la variable de entorno **SERVEROUTPUT** deberá estar en **ON;** en caso contrario los programas no darán ningún error, pero no se visualizará nada.

Para cambiar el estado de la variable introduciremos al comienzo de la sesión:

```
SQL> SET SERVEROUTPUT ON
```

Para pasar datos a un programa podemos recurrir a una de las siguientes opciones:

- Introducir datos en una tabla desde SQL*Plus y, después, leerlos desde el programa.

- Pasar los datos como parámetros en la llamada (en procedimientos y funciones).

- Utilizar variables de sustitución SQL*Plus. Esta opción sólo puede utilizarse con bloques anónimos tal como se muestra en el ejemplo del Caso práctico 5.

8.4 Arquitectura

PL/SQL es una **tecnología integrada** en el servidor Oracle y también en algunas herramientas. Se trata de un motor o gestor que es capaz de ejecutar subprogramas y bloques PL/SQL, y que trabaja en coordinación con el ejecutor de órdenes SQL.

Por defecto, PL/SQL trabaja en modo **interpretado,** es decir, el código almacenado debe ser traducido a código ejecutable cada vez que ejecutamos un procedimiento. A partir de la versión 9iR1 se incluye la **posibilidad de compilación nativa:** los programas se traducen a C en tiempo de compilación y se almacena el código ejecutable. De esta forma los programas se ejecutan hasta 30 veces más rápido.

Con independencia del modo de trabajo, todos los programas PL/SQL son examinados por los analizadores lexicográficos y sintácticos antes de almacenarlos en el servidor (incluso cuando trabajamos en modo interpretado). En este proceso se comprueban también todas las referencias a objetos de la base de datos y se informa de los errores o problemas detectados.

La Figura 8.1 muestra el procesamiento de un bloque PL/SQL. Podemos apreciar que el motor PL/SQL envía las órdenes SQL que contiene el bloque al ejecutor de órdenes SQL, que se encargará de procesarlas.

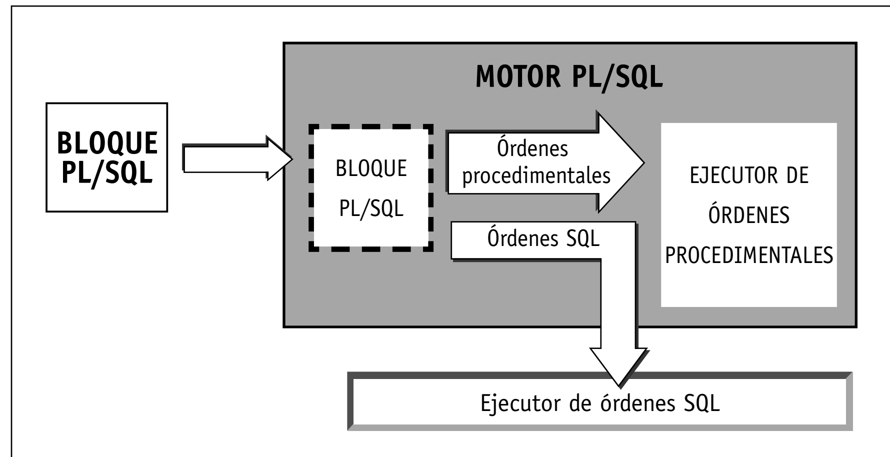


Fig. 8.1 *Procesamiento de un bloque PL/SQL.*

Algunas herramientas de Oracle (Forms, Reports, etcétera) contienen un motor PL/SQL capaz de procesar bloques, PL/SQL ejecutando las órdenes procedimentales en el cliente o en el lugar donde se encuentre la herramienta, enviando al servidor Oracle solamente las instrucciones SQL. En estas cuatro unidades dedicadas a PL/SQL nos ceñiremos exclusivamente al desarrollo de programas PL/SQL para el servidor Oracle.

8.5 Uso y ejemplos de distintos tipos de programas

Para crear nuestros programas podemos utilizar alguna de las herramientas específicas de desarrollo de Oracle, como JDeveloper, que incluyen facilidades gráficas de edición, depuración y compilación. Nosotros hemos optado por utilizar el **entorno SQL*Plus,** pues está disponible en todas las instalaciones Oracle y permite una plena utilización de todas las características del lenguaje.

A. Bloques anónimos

Se pueden generar con diversas herramientas, como SQL*Plus, y se envían al servidor Oracle, donde serán compilados y ejecutados.



SQL*Plus reconoce el comienzo de un bloque anónimo cuando encuentra la palabra reservada DECLARE o BEGIN. Cuando esto ocurre: limpia el buffer SQL, ignora los «;» y entra en modo INPUT.

Las secciones declarativa y de excepciones son opcionales.

```
SQL> BEGIN
     2     DBMS_OUTPUT.PUT_LINE('HOLA MUNDO');
     3 END;
     4 /

    HOLA MUNDO
    Procedimiento PL/SQL terminado con éxito.
```

El ejemplo anterior muestra un bloque anónimo de código que escribe el texto 'HOLA MUNDO'. Para introducirlo se escribe la palabra reservada BEGIN desde el prompt de SQL*Plus seguida de un retorno de carro. Los números de línea se irán mostrando automáticamente. El símbolo «/» provoca que se guarde el bloque en el buffer y lo envía al servidor para su ejecución.

El bloque anterior carece de las secciones declarativas y de gestión de excepciones, pues la única sección obligatoria es la ejecutable. Ésta debe contener, al menos, **un comando ejecutable,** incluso aunque no haga nada tal como aparece en el siguiente ejemplo:

```
SQL> BEGIN
     2     NULL;
     3 END;
     4 /

    Procedimiento PL/SQL terminado con éxito.
```

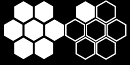
También podemos crear el programa y guardarlo en el buffer SQL pero sin ejecutarlo automáticamente. Para ello se utiliza un punto después de la última línea de código.

El siguiente programa muestra el precio de un producto especificado:

```
SQL> DECLARE
  2    v_precio NUMBER;
  3  BEGIN
  4    SELECT precio_actual INTO v_precio
  5      FROM productos
  6      WHERE PRODUCTO_NO = 70;
  7    DBMS_OUTPUT.PUT_LINE(v_precio);
  8  END;
  9  .
SQL>_
```

Ahora el programa se encuentra cargado en el buffer SQL dispuesto para ser ejecutado utilizando la barra oblicua (/) o el comando **RUN.**

```
SQL> /
450
Procedimiento PL/SQL terminado con éxito.
```





Podemos **guardar** el bloque del buffer en un fichero con la orden **SAVE** según el siguiente formato:

```
SQL > SAVE nombrefichero [REPLACE]
```

La opción REPLACE se usará si el fichero ya existía.

Para **cargar** un bloque de un fichero en el buffer SQL se hará mediante el comando **GET:**

```
SQL> GET nombrefichero
```

Una vez cargado se puede ejecutar tal como acabamos de ver con RUN o con «/».

También se puede **cargar** y **ejecutar** con una sola orden mediante el comando **START:**

```
SQL> START nombrefichero
```

El comando START puede ser sustituido por el símbolo @ con el mismo formato y resultado:

```
SQL> @nombrefichero
```

En PL/SQL podemos insertar comentarios de línea con "--". Todo lo que sigue a estos caracteres dentro de la línea será considerado comentario.

Caso práctico

5 **El siguiente programa solicitará la introducción de un número de cliente y visualizará el nombre del cliente correspondiente con el número introducido. Para introducir el número de cliente recurriremos a las variables de sustitución de SQL*Plus.**

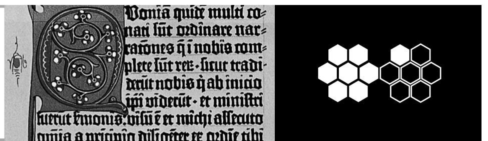
```
SQL> DECLARE
  2      v_nom CLIENTES.NOMBRE%TYPE; --(ejemplo uso %TYPE)
  3     BEGIN
  4       SELECT nombre INTO v_nom
  5         FROM clientes
  6         WHERE CLIENTE_NO=&vn_cli;
  7       DBMS_OUTPUT.PUT_LINE(v_nom);
  8     END;
  9  .
SQL>
```

Para ejecutar el programa utilizaremos RUN o /. Solicitará un valor para la variable de sustitución, y una vez introducido el valor sustituirá a la variable y se enviará el bloque al servidor para su ejecución.

```
SQL> /

Introduzca valor para vn_cli: 102
antiguo   6:       WHERE CLIENTE_NO=&vn_cli;
nuevo     6:       WHERE CLIENTE_NO=102;
LOGITRONICA S.L

   Procedimiento PL/SQL terminado con éxito.
```



Actividades propuestas

1 **Desde el entorno SQL*Plus introduce el bloque del ejemplo anterior, depura los posibles errores y realiza diversas pruebas de ejecución.**

Las variables de sustitución solamente son operativas en los bloques anónimos preferentemente al comienzo y siempre fuera de estructuras condicionales o repetitivas. No podemos usarlas en los procedimientos ya que SQL*Plus realizará la sustitución antes de enviar el bloque al servidor. De esta forma, cuando se compila y almacena el procedimiento se hace con un valor que será siempre el mismo para futuras ejecuciones.

Las variables de sustitución no se pueden usar en procedimientos o funciones.

B. Uso de subprogramas almacenados: procedimientos y funciones

Los **procedimientos** y **funciones** se almacenan en la base de datos, donde quedarán disponibles para ser ejecutados, por ello reciben el nombre de subprogramas almacenados.

Para su creación nos remitimos a lo expuesto en el apartado anterior para los bloques anónimos teniendo en cuenta que, a diferencia de estos, los procedimientos y funciones, una vez compilados, se almacenan en la base de datos y quedan disponibles para su ejecución mediante los comandos apropiados sin necesidad de volver a cargarlos.

Caso práctico

6 **Introduciendo estas líneas desde el indicador de SQL*Plus dispondremos de un procedimiento PL/SQL sencillo para consultar los datos de un cliente:**

```
SQL> CREATE OR REPLACE
PROCEDURE ver_depart (numdepart NUMBER)
AS
   v_dnombre VARCHAR2(14);
   v_localidad VARCHAR2(14);
BEGIN
   SELECT dnombre, loc INTO v_dnombre, v_localidad
     FROM depart
     WHERE dept_no = numdepart;
   DBMS_OUTPUT.PUT_LINE('Num depart:'||numdepart|| ' * Nombre dep:'|| v_dnombre ||
                       ' * Localidad:'||v_localidad);
EXCEPTION
   WHEN NO_DATA_FOUND THEN
   DBMS_OUTPUT.PUT_LINE('No encontrado departamento ');
END ver_depart;
/

Procedimiento creado.
```

En el ejemplo anterior podemos observar:

- La **cabecera del procedimiento** (línea 2) indica el nombre del mismo y el parámetro que se utilizará para pasarle valores.

- La palabra **AS** especifica el comienzo del cuerpo del programa y del bloque, comenzando con la zona de declaraciones donde se indicarán los objetos que intervendrán en el programa: variables, constantes, etcétera.

- La palabra **BEGIN** indica el comienzo de la zona de instrucciones.

- La instrucción **SELECT** deposita el resultado de la consulta en las variables *v_dnombre, v_localidad* que siguen a INTO tal como comentamos al hablar del soporte para consultas del lenguaje. La cláusula WHERE contiene el parámetro con el que se llamó al programa, que será el número de departamento cuyos datos se requieren.

- El procedimiento DBMS_OUTPUT.PUT_LINE visualiza el contenido de las variables y los literales. Para que funcione correctamente la variable SERVEROUTPUT debe estar en ON antes de ejecutar el programa.

- La palabra EXCEPTION indica el comienzo de la zona de tratamiento de excepciones. Esta zona sólo se ejecutará si se produzce alguna excepción. Por ejemplo, si al ejecutarse la cláusula SELECT no se encuentra ninguna fila que cumpla la condición, se levantará la excepción NO_DATA_FOUND, se bifurcará el control del programa a esta sección.

- El manejador WHEN NO_DATA_FOUND «caza» la excepción, en el caso de que se produzca, y ejecuta las instrucciones que siguen a la cláusula THEN dando por finalizado el programa.

- El símbolo «/» indica a SQL*Plus el final del procedimiento. Es una abreviatura del comando RUN. Compila y guarda en la base de datos el programa introducido.

  Si el compilador detecta errores veremos el siguiente mensaje:

  ```
  Aviso: Procedimiento creado con errores de compilación.
  ```

La orden **SHOW ERRORS** permite ver los errores detectados. Invocaremos al editor y subsanaremos el error. Hecho esto el programa se compilará y almacenará de nuevo, introduciendo «/» seguido de INTRO.

Ahora, el procedimiento se encuentra almacenado en el servidor de la base de datos y puede ser invocado desde cualquier estación por cualquier usuario autorizado y con cualquier herramienta; por ejemplo, desde SQL*Plus mediante la orden EXECUTE:

```
SQL> SET SERVEROUTPUT ON
SQL>  EXECUTE ver_depart(20)
Num depart:20 * Nombre dep: INVESTIGACION * Localidad:
MADRID
```

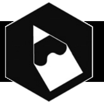


También podemos invocar al procedimiento desde un bloque PL/SQL, por ejemplo de esta forma:

```
SQL> BEGIN ver_depart(30); END;
  /
Num depart:30 * Nombre dep:VENTAS * Localidad: BARCELONA
```

En realidad, la orden EXECUTE es una utilidad de SQL*Plus que crea un bloque PL/SQL anónimo añadiendo un BEGIN y un END al principio y al final, respectivamente, de la llamada al procedimiento.

Caso práctico



7 Ejemplos de aplicación:

1. En el siguiente procedimiento se visualiza el precio de un producto cuyo número se pasa como parámetro.

```
SQL> CREATE OR REPLACE
        PROCEDURE ver_precio(v_num_producto NUMBER)
     AS
        v_precio NUMBER;
     BEGIN
        SELECT precio_actual INTO v_precio
          FROM productos
          WHERE producto_no = v_num_producto;
        DBMS_OUTPUT.PUT_LINE('Precio = '||v_precio);
      END;
      /

Procedimiento creado.

SQL> EXECUTE VER_PRECIO(50);
   Precio = 1050

  Procedimiento PL/SQL terminado con éxito.
```

2. Escribiremos un procedimiento que modifique el precio de un producto pasándole el número del producto y el nuevo precio. El procedimiento comprobará que la variación de precio no supere el 20 por 100:

```
SQL> CREATE OR REPLACE
     PROCEDURE modificar_precio_producto
        (numproducto NUMBER, nuevoprecio NUMBER)
     AS
       v_precioant NUMBER(5);
     BEGIN
       SELECT precio_actual INTO v_precioant
           FROM productos
             WHERE producto_no = numproducto;
```

(Continúa)

(Continuación)

```
   IF (v_precioant * 0.20) > (nuevoprecio - v_precioant) THEN
       UPDATE productos SET precio_actual = nuevoprecio
         WHERE producto_no = numproducto;
   ELSE
         DBMS_OUTPUT.PUT_LINE('Error, modificación supera 20%');
   END IF;

EXCEPTION
   WHEN NO_DATA_FOUND THEN
   DBMS_OUTPUT.PUT_LINE('No encontrado producto '|| numproducto);
END modificar_precio_producto;
/
```

Ejemplos de ejecución:

```
SQL> SET SERVEROUTPUT ON

SQL>  EXECUTE MODIFICAR_PRECIO_PRODUCTO(60,300)
Procedimiento PL/SQL terminado con éxito.

SQL>  SELECT PRECIO_UNI FROM PRODUCTOS WHERE COD_PRODUCTO=60;
PRECIO_UNI
----------
      300

SQL> EXECUTE MODIFICAR_PRECIO_PRODUCTO(60,10000)
Error, modificación supera 20%
Procedimiento PL/SQL terminado con éxito.

SQL>  SELECT PRECIO_UNI FROM PRODUCTOS WHERE COD_PRODUCTO=3;
PRECIO_UNI
----------
       300
```

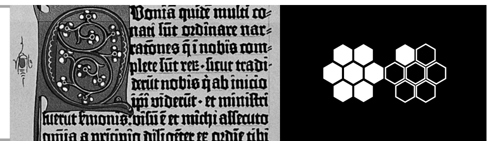
Observamos que en el segundo caso no se ha producido la modificación deseada. Aun así, el procedimiento ha terminado con éxito, ya que no se ha producido ningún error no tratado.

3. Escribiremos una función que devuelva el valor con IVA de una cantidad que se pasará como primer parámetro. La función también podrá recoger un segundo parámetro opcional, que será el tipo de IVA siendo el valor por defecto 16.

```
SQL> CREATE OR REPLACE FUNCTION con_iva (
       cantidad NUMBER,
       tipo NUMBER DEFAULT 16)

       RETURN NUMBER
   AS
       v_resultado NUMBER (10,2) DEFAULT 0;
   BEGIN
```

(Continúa)



(Continuación)

```
    v_resultado := cantidad * (1 + (tipo / 100));
    RETURN(v_resultado);
END con_iva;
/

Función creada.
```

Ahora podemos usar la función creada para realizar cálculos dentro de un bloque o programa PL/SQL:

```
SQL>  BEGIN  DBMS_OUTPUT.PUT_LINE(con_iva(200)); END;
  /
232
```

Debemos recordar que hay que HACER ALGO con el valor devuelto por la función: visualizarlo, usarlo como parte de una expresión, etcétera. También podemos usarla en instrucciones SQL:

```
SQL> SELECT producto_no, precio_actual, con_iva(precio_actual) FROM productos;

   PRODUCTO_NO    PRECIO_ACTUAL    CON_IVA(PRECIO_ACTUAL)
   -----------    -------------    ----------------------
       10              550                 638
       20              670                 777,2
       30              460                 533,6
       40              340                 394,4
       50             1050                1218
       60              280                 324,8
       70              450                 522
       80              550                 638
```

En los ejemplos anteriores se muestra el código tal como aparece en el entorno SQL*Plus, con los números de línea y el indicador SQL.

En lo sucesivo prescindiremos de esos elementos para centrarnos en el código.

Conceptos básicos



- PL/SQL es un lenguaje de programación que soporta:
  - Instrucciones SQL de consulta y manipulación de datos.
  - Estructuras de control y otros elementos de lenguajes de tercera generación.
  - Programación Orientada a Objetos.

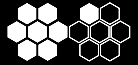
- El bloque es la unidad básica de todos los programas PL/SQL. Su formato básico incluye tres secciones:
  - [**DECLARE** - Sección declarativa: se declaran las variables y otros objetos
    `<declaraciones>`]
  - **BEGIN** -- Sección ejecutable: se incluyen las instrucciones del programa.
    `<órdenes>`
  - [**EXCEPTION** -- Sección de excepciones: se tratan las posibles excepciones
    `<gestión de excepciones>`]
  - **END;**

- En PL/SQL podemos distinguir los siguientes tipos de programas:
  - **Bloques anónimos.** Sin nombre. Se ejecutan en el servidor pero no se guardan.
  - **Subprogramas:** *procedimientos* y *funciones*. Se guardan y ejecutan en el servidor.
  - **Disparadores de base de datos.** Se ejecutan al producirse un evento en la BD.

- Para crear y ejecutar bloques anónimos usaremos:
  - Para crear un bloque desde SQL*Plus simplemente introduciremos las palabras reservadas DECLARE o BEGIN y, a continuación, el resto del bloque.
  - Para guardar un bloque en un fichero utilizaremos el comando `SAVE` *`nombredefichero`*.
  - Para cargar un bloque guardado en un fichero usaremos `GET` *`nombredefichero`*.
  - Para ejecutar un bloque cargado en memoria usaremos `RUN` o «/».
  - Para cargar y ejecutar un fichero que contiene un bloque usaremos `START` *`nobrefichero`* o @*`nombrefichero`*.
  - Para visualizar los errores que puedan producirse durante la compilación usaremos SHOW ERRORS.

- Para visualizar datos desde un programa PL/SQL usaremos DBMS_OUTPUT.PUT_LINE. Al ejecutar el programa debemos asegurarnos de que la variable del sistema SEVEROUTPUT está en ON (SET SERVEROUTPUT ON).





Actividades complementarias



1. **Construye un bloque PL/SQL que escriba el texto 'Hola'.**

2. **¿Qué hace el siguiente bloque PL/SQL?**

```
SQL>DECLARE
      v_num NUMBER;
    BEGIN
      SELECT count(*) INTO v_num
        FROM productos;
      DBMS_OUTPUT.PUT_LINE(v_num);
    END;
    /
```

3. **Introduce el bloque anterior desde SQL*Plus y guardarlo en un fichero.**

4. **Ejecuta la orden SELECT especificada en el bloque anterior desde SQL*Plus sin la cláusula INTO.**

5. **Carga y ejecuta el bloque de nuevo, y comprueba que el resultado aparece en pantalla.**

6. **Escribe desde SQL*Plus el ejemplo número 1 del epígrafe «Uso de subprogramas almacenados» y prueba a ejecutarlo con distintos valores.**

7. **Identifica en el ejemplo número 2 del epígrafe «Uso de subprogramas almacenados» los siguientes elementos:**

   – La cabecera del procedimiento.

   – El nombre del procedimiento.

   – Los parámetros del procedimiento.

   – Las variables locales.

   – El comienzo y el final del bloque PL/SQL.

   – El comienzo y el final de las secciones declarativa, ejecutable y de gestión de excepciones.

   – ¿Qué hace la cláusula INTO?

   – ¿Qué hace WHEN NO_DATA_FOUND?

   – ¿Por qué no tiene la cláusula DECLARE? ¿Qué tiene en su lugar?

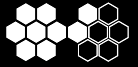
9

Fundamentos del lenguaje PL/SQL

En esta unidad aprenderás a:

1. **Declarar variables y otros objetos empleando los tipos de datos disponibles.**
2. **Manejar operadores, funciones predefinidas y otros elementos del lenguaje.**
3. **Controlar el flujo de ejecución de nuestros programas.**
4. **Realizar procedimientos y funciones para desarrollar programas.**
5. **Usar parámetros de distintos tipos.**





9.1 Introducción

En la unidad anterior hemos estudiado las características generales de PL/SQL y su utilización desde una perspectiva global y un tanto intuitiva. En esta unidad nos ocuparemos de los principales elementos del lenguaje PL/SQL: tipos de datos, variables, operadores y expresiones, reglas sintácticas, estructuras de control, así como de su uso en procedimientos y funciones.

9.2 Tipos de datos

PL/SQL dispone de los mismos tipos de datos que SQL, además de otros propios. En la mayoría de los casos existe compatibilidad con los tipos correspondientes soportados por la base de datos pero también existen diferencias que deben tenerse en cuenta.

Podemos clasificar los tipos de datos soportados por PL/SQL en las siguientes categorías:

- **Escalares:** almacenan valores simples. A su vez pueden subdividirse en:
  - *Carácter/Cadena*: CHAR, NCHAR, VARCHAR2, NVARCHAR2, LONG, RAW, LONG RAW, ROWID, UROWID.
  - *Numérico*: NUMBER, BINARY_INTEGER, PLS_INTEGER, BINARY_DOUBLE, BINARY_FLOAT.
  - *Booleano*.
  - *Fecha/hora*: DATE, TIMESTAMP, INTERVAL.
  - *Otros*: ROWID, UROWID.
- **Compuestos:** Son tipos compuestos de otros simples. Los veremos en la unidad de programación avanzada. Por ejemplo:
  - *Tablas indexadas*.
  - *Tablas anidadas*.
  - *Varrays*.
  - *Objetos*.
- **Referencias:** Difieren de los demás por sus características de manejo y almacenamiento.
  - *REF CURSOR:* Son referencias a cursores.
  - *REF:* Son punteros a objetos.

En PL/SQL el programador puede definir sus propios tipos a partir de los anteriores.

- **LOB:** Almacenan objetos de grandes dimensiones (*Large OBject*) de hasta 128 terabytes a partir de la versión 10gR1. Sustituyen con ventaja a los tipos LONG y LONG RAW.

A continuación, trataremos los datos escalares más utilizados, dejando para la de programación avanzada los datos compuestos, referencias y objetos.

A. Tipos escalares

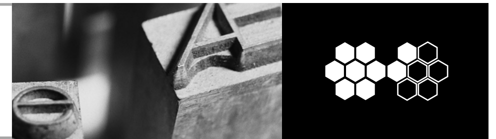
En las siguientes tablas podemos observar los distintos tipos de datos escalares soportados por PL/SQL y sus descripciones.

CARÁCTER
Almacena **cadenas** de caracteres cuya longitud máxima es 32 KBytes. Al especificar las longitudes debemos tener en cuenta que, por defecto, cada carácter ocupa 1 byte, aunque hay juegos de caracteres que ocupan más (por ejemplo, el asiático).

CHAR[(L)]
 	
NCHAR[(L)]
	Almacena cadenas de caracteres de **longitud fija.** Opcionalmente se puede especificar, entre paréntesis, la longitud máxima; en caso contrario, el valor por defecto es 1 y no se pone el paréntesis. Las posiciones no utilizadas **se rellenan con blancos.**
	Ejemplo:
	`Nombre CHAR(15);`
	`Letra_nif CHAR;`
	`Situacion CHAR();--ERROR`
	
	La longitud de **CHAR** se expresa por defecto en **bytes** pero se puede expresar en caracteres del juego de caracteres nacional utilizando la opción CHAR:
	
	`Apellido CHAR(25 CHAR);`
	
	La longitud en **NCHAR** se expresa siempre en caracteres del juego de **caracteres** nacional. Equivale a usar la opción CHAR comentada arriba:
	
	`Apellido NCHAR(25); -- equivalente a la anterior`

VARCHAR2(L)
 	
NVARCHAR2(L)
 	
VARCHAR(L)
	Almacena cadenas de caracteres de **longitud variable** cuyo límite máximo se debe especificar.
	Ejemplos:
	`Apellidos VARCHAR2(25);`
	`Control VARCHAR2; --ERR falta la longitud`
	`Nombre VARCHAR2(15 CHAR);`
	`Nombre NVARCHAR2(25);`
	
	También está disponible el tipo VARCHAR por compatibilidad con el estándar SQL, aunque Oracle desaconseja su utilización recomendando utilizar en su lugar VARCHAR2 y NVARCHAR2 para beneficiarse de la evolución de estos tipos.

LONG[(L)]	Almacena cadenas de longitud variable. Es similar a VARCHAR2 pero no permite la opción CHAR.
	Ejemplos:
	`v_Observaciones LONG(5000);`
	`Resumen LONG;`
	
	Debemos tener en cuenta que el límite para este tipo en variables PL/SQL es de 32 KB, pero las columnas de este tipo admiten hasta 2GB.



NUMÉRICOS	

NUMBER(P, E)	Donde *P* es la precisión (número total de dígitos) y *E* es la escala (número de decimales).
	La precisión y la escala son opcionales, pero si se especifica la escala hay que indicar la precisión.
	Ejemplo:
	`Importe NUMBER(5,2); -- admite hasta 999.99`
	`Centimos NUMBER(,2); -- ERROR lo correcto es`
	`Centimos NUMBER(2,2); -- admite hasta .99`
	
	Si al asignar un valor se excede la escala, se producirá *truncamiento*. Si se excede la precisión, se producirá un *error*.
	Ejemplo:
	`Importe:= 1000; -- ERROR excede la escala`
	`Importe:= 234.344; -- guarda 234.34`
	
	PL/SQL dispone de subtipos de NUMBER que se utilizan por compatibilidad y/o para establecer restricciones. Son DECIMAL, NUMERIC, INTEGER, REAL, SMALLINT, etcétera.

BINARY_INTEGER	Es un tipo numérico entero que se almacena en memoria en formato binario para facilitar los cálculos. Puede contener valores comprendidos entre –2147483647 y + 2147483647. Se utiliza en contadores, índices, etcétera. Por ejemplo: `Contador BINARY_INTEGER;` Subtipos: NATURAL y POSITIVE.
PLS_INTEGER	Similar a BINARY_INTEGER y con el mismo rango de valores, pero tiene dos ventajas respecto al anterior: • Es más rápido. • Si se da desbordamiento en el cálculo se produce un error y se levanta la excepción correspondiente, lo cual no ocurre con BINARY_INTEGER. Por ejemplo: `indice PLS_INTEGER;`
BINARY_DOUBLE BINARY_FLOAT	Disponibles desde la versión 10gR1, su utilización se circunscribe al ámbito científico para realizar cálculos muy precisos y complejos conforme a la norma IEEE 754. Oracle no los soporta directamente como columnas en la base de datos. No los utilizaremos en este libro. Remitimos al lector a la documentación de Oracle para su eventual utilización.

BOOLEANO	

BOOLEAN	Almacena valores lógicos TRUE, FALSE y NULL. Para representar estos valores **no debemos utilizar comillas.**
	Ejemplo:
	`v_ocupado BOOLEAN;`
	`v_eliminado BOOLEAN DEFAUL FALSE;`
	
	La base de datos no soporta este tipo para definir columnas.

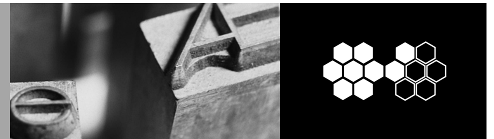
FECHA / HORA Almacenan valores de tiempo. Son idénticos a los tipos correspondientes de la base de datos.	
DATE **TIMESTAMP** [(P)] **TIMESTAMP** [(P)] **WITH TIME ZONE**	Almacena fechas incluyendo la hora. Los formatos en que muestran la información son los establecidos por defecto, aunque se pueden especificar máscaras de formato. Por defecto, sólo se muestra la fecha, pero también se almacena la hora: `día/mes/año horas:minutos:segundos.` **TIMESTAMP** también almacena fecha y hora pero incluyendo fracciones de segundo. La precisión de estas fracciones se puede especificar como parámetro (por defecto es 6, es decir, 999999 millonésimas de segundo, pero puede variar entre 0 y 9). Normalmente el valor se toma de la variable del sistema SYSTIMESTAMP. **TIMESTAMP WITH TIME ZONE** incluye el valor de la **zona horaria.**

TIMESTAMP [(P)] **WITH LOCAL TIME ZONE**	**TIMESTAMP WITH LOCAL TIME ZONE** igual que TIMESTAMP, pero:
	– Los datos **se almacenan normalizados a la zona horaria donde se encuentra la base de datos.**
	– Los datos recuperados **se muestran en el formato de la zona horaria correspondiente a la sesión** de los usuarios.
	
	El siguiente ejemplo ilustra estos conceptos:
	
	`DECLARE`
	`fechaTI TIMESTAMP;`
	`fechaTZ TIMESTAMP WITH TIME ZONE;`
	`fechaTL TIMESTAMP WITH LOCAL TIME ZONE;`
	`BEGIN /* incluiremos dos instrucciones en la misma linea */`
	`fechaTI := SYSTIMESTAMP;   DBMS_OUTPUT.PUT_LINE('fechaTI:'‖fechaTI);`
	`fechaTZ := SYSTIMESTAMP;   DBMS_OUTPUT.PUT_LINE('fechaTZ:'‖fechaTZ);`
	`fechaTL := SYSTIMESTAMP;   DBMS_OUTPUT.PUT_LINE('fechaTL:'‖fechaTL);`
	`END;`
	
	El resultado de la ejecución será:
	
	`fechaTI:02/04/06 10:07:40,196000`
	`fechaTZ:02/04/06 10:07:40,196000 +02:00`
	`fechaTL:02/04/06 10:07:40,196000`
	
	En este caso, FechaTL tiene el mismo valor que FechaTI porque la zona horaria de almacenamiento y recuperación es la misma.

INTERVAL YEAR [(PY)] TO MONTH
 	
INTERVAL DAY [(PD)] TO SECOND [(PS)]
	Los subtipos de INTERVAL representan intervalos de tiempo entre dos fechas. La diferencia entre ellos es la manera de expresar el intervalo:
	
	– INTERVAL YEAR TO MONTH se expresa en años/meses. Opcionalmente se puede incluir la precisión para el número de años, entre 0 y 9 dígitos, por defecto es 2.
	
	`v_anios_meses2 INTERVAL YEAR TO MONTH := '1-06';`
	
	– INTERVAL DAY TO SECOND se expresa en días/segundos. Opcionalmente se puede incluir la precisión para el número de días (entre 0 y 9 dígitos, por defecto es 2), y para el número de segundos (entre 0 y 9 dígitos, por defecto es 6).
	
	`v_dias_segundos INTERVAL DAY TO SECOND DEFAULT '02 14:0:0.0';`
	
	Estos tipos se utilizan cuando se requiere manejar diferencias de tiempo con precisiones expresadas en fracciones de segundo. En la mayoría de los casos recurriremos a los tipos tradicionales junto con las funciones disponibles MONTHS_BETWEEN, etcétera.

OTROS TIPOS ESCALARES	
RAW(L)	Almacena datos binarios en longitud fija. Se utiliza para almacenar cadenas de caracteres evitando problemas por las conversiones entre conjuntos de caracteres que realiza Oracle.
LONG RAW	Almacena datos binarios en longitud variable evitando conversiones entre conjuntos de caracteres.
ROWID **UROWID**[(L)]	Almacenan identificadores de direcciones de fila. Son idénticos a los tipos correspondientes de la base de datos. UROWID permite especificar la longitud de almacenamiento en número de bytes (4000 bytes es el máximo y el valor por defecto).

Tabla 9.1. *Tipos de datos escalares en PL/SQL.*



B. Principales diferencias de almacenamiento con la base de datos Oracle

Aunque PL/SQL ofrece soporte a los tipos de datos de las columnas de la base de datos Oracle, en algunos casos, no existe una equivalencia exacta entre los tipos de PL/SQL y los mismos tipos de las columnas de Oracle, especialmente en lo relativo a las longitudes máximas que pueden almacenar. La siguiente tabla revela las diferencias más significativas.

Oracle realiza conversiones automáticas de tipos de datos (de manera acertada en muchos casos). No obstante, es preferible especificar las conversiones entre tipos empleando las funciones estudiadas en SQL.

DIFERENCIAS DE ALMACENAMIENTO ENTRE PL/SQL Y LAS COLUMNAS DE LA BASE DE DATOS ORACLE

TIPO	VARIABLES PL/SQL	COLUMNAS DB ORACLE
VARCHAR2 NVARCHAR2	32.767 bytes	4000 bytes
CHAR NCHAR	32.767 bytes	4000 bytes
LONG	32.767 bytes	2 GB
RAW	32.767 bytes	2.000 bytes
LONG RAW	32.767 bytes	2 GB
BOOLEAN	DISPONIBLE	NO SOPORTADO

9.3 Identificadores

Los **identificadores** se utilizan para nombrar los objetos que intervienen en un programa: variables, constantes, cursores, excepciones, procedimientos, funciones, etiquetas, etcétera.

En PL/SQL deben cumplir las siguientes características:

Se pueden saltar algunas de estas reglas utilizando identificadores entre comillas dobles (por ejemplo `"2ªvaria ble/"`) pero no es aconsejable.

- Pueden tener entre 1 y 30 caracteres de longitud.
- El primer carácter debe ser una letra.
- Los restantes caracteres deben ser caracteres alfanuméricos o signos admitidos (letras, dígitos, los signos de dólar, almohadilla y subguión).
- No pueden incluir signos de puntuación, espacios, etcétera.

Ejemplos de identificadores válidos	`v_`	`A#$`	`X2`	`anio`	`v_num.`
Ejemplos de identificadores no válidos	`_v`	`#A`	`2X`	`año`	`v-num.`

PL/SQL no diferencia entre mayúsculas y minúsculas en los identificadores ni en las palabras reservadas. Por ejemplo, podemos escribir `V_Num_Meses`, `v_num_meses` o `V_NUM_MESES`. En los tres casos se trata del mismo identificador.

9.4 Variables

Las **variables** sirven para almacenar información cuyo valor puede cambiar a lo largo de la ejecución del programa.

A. Declaración e inicialización de variables

Todas las variables PL/SQL deben declararse en la *sección declarativa* antes de su uso. El formato genérico para declarar una variable es el siguiente:

```
<nombre_de_variable> <tipo> [NOT NULL] [{:= | DEFAULT}
                        <valor>];
```

La opción **DEFAULT** (o bien la asignación «:=») sirve para asignar valores por defecto a la variable desde el momento de su creación.

```
DECLARE
     importe NUMBER(8,2);
     contador NUMBER(2,0) := 0;
     nombre CHAR(20) NOT NULL := "MIGUEL";
...
```

Para cada variable se debe especificar el *tipo*. No se puede, como en otros lenguajes, indicar una lista de variables del mismo tipo y, a continuación, el tipo. PL/SQL no lo permite.

La opción **NOT NULL** fuerza a que la variable tenga siempre un valor. Si se usa, deberá inicializarse la variable en la declaración con DEFAULT o con := para la inicialización. Por ejemplo:

```
nombre CHAR(20) NOT NULL DEFAULT "MIGUEL";
```

También se puede inicializar una variable que no tenga la opción NOT NULL. Por ejemplo:

```
nombre CHAR(20) DEFAULT "MIGUEL";
```

Si no se inicializan las variables en PL/SQL, se garantiza que su valor es NULL.

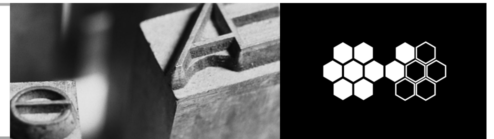


Actividades propuestas

1 **Indica los errores que aparecen en las siguientes instrucciones y la forma de corregirlos.**

```
DECLARE
   Num1 NUMBER(8,2) := 0
     Num2 NUMBER(8,2) NOT NULL := 0;
```

(Continúa)



(Continuación)

```
    Num3 NUMBER(8,2) NOT NULL;
    Cantidad INTEGER(3);
    Precio, Descuento NUMBER(6);
    Num4 Num1%ROWTYPE;
    Dto CONSTANT INTEGER;
  BEGIN
    ...
  END;
```

B. Uso de los atributos %TYPE y %ROWTYPE

En lugar de indicar explícitamente el tipo y la longitud de una variable existe la posibilidad de utilizar los atributos %TYPE y %ROWTYPE para declarar variables que sean del mismo tipo que otros objetos ya definidos.

– **%TYPE** declara una variable del mismo tipo que otra, o que una columna de una tabla. Su formato es:

```
nombre_variable  nombre_objeto%TYPE;
```

– **%ROWTYPE** declara una variable de registro cuyos campos se corresponden con las columnas de una tabla o vista de la base de datos. Su formato es:

```
nombre_variable  nombre_objeto%ROWTYPE;
```

Ejemplos:

- `total importe%TYPE;`

  Declara la variable `total` del mismo tipo que la variable `importe` que se habrá definido previamente.

- `nombre_moroso clientes.nombre%TYPE;`

  Declara la variable `nombre_moroso` del mismo tipo que la columna `nombre` de la tabla `clientes`.

- `moroso clientes%ROWTYPE;`

  Declara la variable `moroso` que podrá contener una fila de la tabla `clientes`.

Al declarar una variable del mismo tipo que otro objeto usando los atributos %TYPE y %ROWTYPE, se hereda el tipo y la longitud, pero no los posibles atributos NOT NULL ni los valores por defecto que tuviese definidos el objeto original.

Para hacer referencia a cada uno de los campos indicaremos el nombre de la variable, un punto y el nombre del campo que coincide con el de la columna correspondiente. Por ejemplo:

```
DBMS_OUTPUT.PUT_LINE(moroso.nombre);
```

C. Ámbito y visibilidad de las variables

Las **variables** se crean al comienzo del bloque y dejan de existir una vez finalizada la ejecución del bloque en el que han sido declaradas. El ámbito de una variable incluye el bloque en el que se declara y sus bloques «hijos».

Una variable declarada en un bloque será **local** para el bloque en el que ha sido declarada y **global** para los bloques hijos de éste. Las variables declaradas en los bloques hijos no son accesibles desde el bloque padre.

En el siguiente ejemplo, podemos observar que `v1` es accesible para los dos bloques (es local al bloque padre y global para el bloque hijo), mientras que `v2` solamente es accesible para el bloque hijo.

```
DECLARE      -------------------------Bloque PADRE
   v1 CHAR;
BEGIN
   v1 := 27;

   DECLARE  ------------------Bloque HIJO
      v2 CHAR;
   BEGIN
      v2:= 2;
      v1:= v2;
      ...
   END;  ---------------------Fin bloque HIJO

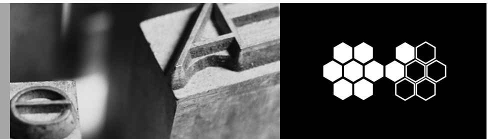
   v2:= v1; --> Error: v2 no existe en este ámbito.

END;  --------------------------------Fin bloque PADRE
```

En el caso de que un identificador local coincida con uno global, si no se indica más, se referencia el local. Es decir, el identificador local dentro de su ámbito oculta la visibilidad del global. No obstante, se pueden utilizar etiquetas y cualificadores para deshacer ambigüedades.

```
<<padre>>    -- etiqueta que identifica al bloque padre.
DECLARE
   v CHAR;
BEGIN
   ...
   DECLARE
        v CHAR;
   BEGIN
        ...
        v := padre.v;-- el primero es el identificador local
        ...
   END;
END;
```

En caso de coincidencia, los identificadores de columnas tienen precedencia sobre las variables y parámetros formales; éstos, a su vez, tienen precedencia sobre los nombres de tablas.



9.5 Constantes y literales

Para representar valores que no variarán a lo largo de la ejecución del programa disponemos de dos elementos: *constantes* y *literales*.

Aunque PL/SQL no diferencia entre mayúsculas y minúsculas en los identificadores, sí lo hace en los literales y con el contenido de las variables y constantes (tal como ocurre en SQL).

A. Constantes

Las declaramos con el siguiente formato:

```
<nombre_de_constante> CONSTANT <tipo> := <valor>;
```

Cuando declaramos una constante siempre se deberá asignar un valor. Por ejemplo:

```
Pct_iva CONSTANT REAL := 16;
```

B. Literales

Los **literales** representan valores constantes directamente (sin recurrir a identificadores). Se utilizan para visualizar valores, hacer asignaciones a variables, constantes u otros objetos del programa. Pueden ser de varios tipos:

Carácter

Constan de un único carácter alfanumérico o especial introducido entre comillas simples. Ejemplos : `'A', 'a', 'j', '5', '%', '*'`

Cadena

Son conjuntos de caracteres introducidos entre comillas simples. Ejemplos: `'Hola Mundo', 'Cliente Nº: ', 'Introduzca un valor...'`

Debemos ser muy cuidadosos con estos delimitadores, especialmente si utilizamos editores distintos del SQL*Plus pues muchos editores inducen a confusión o tienden a cambiar estos signos.

En ocasiones necesitaremos incluir el carácter utilizado como delimitador en la cadena o carácter que queremos representar. Por ejemplo: `DBMS_OUTPUT.PUT_LINE('Peter's hotel');`.

Dará el error: `ORA-01756: quoted string not properly terminated.`

En estos casos emplearemos dos comillas simples en el lugar donde debe aparecer una:

```
DBMS_OUTPUT.PUT_LINE('Peter''s hotel'); END;
```

El resultado será el deseado: `Peter's hotel.`

Numérico

Representan valores numéricos enteros o reales. Se pueden representar con notación científica. Ejemplos: `4, -7, 567.45, -458.456, 2.234E4, 5.25e-5, -8,75E+2`

Booleano

Representan los valores lógicos verdadero, falso y nulo. Se representan mediante las palabras **TRUE, FALSE** y **NULL** escritas directamente en el código y sin comillas (no son literales sino palabras reservadas del lenguaje). Se pueden utilizar para asignar valores a variables y constantes, o para comprobar el resultado de expresiones lógicas. Ejemplos: `v_cobrado := TRUE;  v_enviado := FALSE;`

Fecha/hora

Se utilizan para representar valores de fecha y hora según los distintos formatos estudiados anteriormente. Se representan mediante la expresión que indica el tipo DATE o TIMESTAMP seguida de la cadena delimitada que indica el valor que queremos representar. Por ejemplo: `DATE '2005-11-09'   TIMESTAMP '2005-11-09 13:50:00'`.

También podemos representar valores correspondientes a los **subtipos TIMESTAMP WITH TIME ZONE** y **TIMESTAMP WITH LOCAL TIME ZONE** utilizando la palabra TIMESTAMP y la cadena en el formato correspondiente al subtipo que queremos representar como se muestra en los siguientes ejemplos:

```
TIMESTAMP '2005-11-09 13:50:00 +06:00' (para TIMESTAMP WITH
TIME ZONE)

TIMESTAMP '2005-11-09 13:50:00' (para TIMESTAMP WITH LOCAL
TIME ZONE)
```

9.6 Operadores y delimitadores

Los operadores se utilizan para asignar valores y formar expresiones. PL/SQL dispone de operadores de:

- **Asignación.** **:=** Asigna un valor a una variable.
  Por ejemplo: `Edad := 19;`

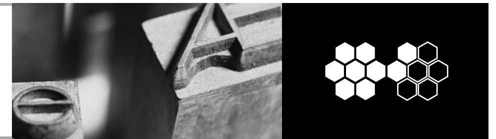
- **Concatenación.** || Une dos o más cadenas.
  Por ejemplo: `'buenos' || 'días'` dará como resultado `'buenosdías'`.

- **Comparación.** `=, !=, <, >, <=, >=, IN, IS NULL, LIKE, BETWEEN,`...
  Funcionan igual que en SQL.

- **Aritméticos.** +, -, *, /, **. Se emplean para realizar cálculos. Algunos de ellos se pueden utilizar también con fechas.

  f1 - f2. Devuelve el número de días que hay entre las fechas *f1* y *f2*.

  f + n. Devuelve una fecha que es el resultado de sumar *n* días a la fecha *f*.



f - *n*. Devuelve una fecha que es el resultado de restar *n* días a la fecha *f*.

– **Lógicos. AND, OR** y **NOT.** Permiten operar con expresiones que devuelven valores booleanos (comparaciones, etcétera). A continuación, se detallan las tablas de valores de los operadores lógicos AND, OR y NOT, teniendo en cuenta todos los posibles valores, incluida la ausencia de valor (NULL).

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

Tabla 9.2. *Valores de los operadores lógicos.*

– **Otros indicadores y delimitadores.**

( )	Delimitador de expresiones.
' '	Delimitador de literales de cadena.
" "	Delimitador de identificadores (utilización desaconsejable, en general).
<< >>	Etiquetas.
/* */	Delimitador de comentarios de varias líneas.
--	Indicador de comentario de una línea.
%	Indicador de atributo (TYPE, ROWTYPE, FOUND,...).
:	Indicador de variables de transferencia (*bind*).
,	Separador de ítem de lista.
;	Terminador de instrucción.
@	Indicador de enlace de base de datos.

A. Orden de precedencia en los operadores

El **orden de precedencia** o **prioridad de los operadores** determina el orden de evaluación de los operandos de una expresión. Por ejemplo, la siguiente expresión: 12+18/6, dará como resultado 15, ya que la división se realizará primero. En la siguiente tabla se muestran los operadores disponibles agrupados y ordenados de mayor a menor, por orden de precedencia.

Prioridad	Operador	Operación
1	**, NOT	Exponenciación, negación.
2	*, /	Multiplicación, división.
3	+, -, ||	Suma, resta, concatenación.
4	=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparación.
5	AND	Conjunción.
6	OR	Disyunción.

Tabla 9.3. *Orden de precedencia en los operadores.*

Debemos tener cuidado con expresiones del tipo:

```
100 > x > 0  -- ilegal
```

Ya que primero se ejecutará 100 > x y devolverá un valor que será VERDADERO, FALSO o NULL, y ése será el primer término de la siguiente comparación, lo cual causará un error. Para evitarlo, podemos indicar la expresión de la siguiente forma:

```
(100 > x) AND (x > 0)
```

Aunque ésta es la prioridad establecida por defecto, podemos cambiarla utilizando paréntesis. Por ejemplo, en la expresión anterior, si queremos que la suma se haga antes que la división, lo indicaremos con la expresión: (12+18)/6. El resultado será 5.

Los operadores que se encuentran en el mismo grupo tienen la misma precedencia. En estos casos no se garantiza el orden de evaluación. Si queremos que se evalúen en algún orden concreto, deberemos utilizar paréntesis.

B. Evaluación en cortocircuito

PL/SQL da por concluida la evaluación de una expresión lógica cuando el resultado ha quedado determinado por la evaluación de una parte de ella. Esto ocurre cuando en una operación OR el primer operando es verdadero, o en una operación AND el primer operando es falso. De esta forma, los programas se ejecutan más rápidamente.

La evaluación en cortocircuito se puede aprovechar para evitar algunos problemas. Por ejemplo, suponemos que `x` e `y` son variables. Si queremos comprobar que `y / x` es mayor que 1, podemos evitar la posibilidad de un error ORA-01476 (intento de división por cero) aprovechando la evaluación en cortocircuito:

```
IF (x != 0) AND  (y / x > 1) THEN ...
```

9.7 Funciones predefinidas

En PL/SQL se pueden utilizar todas las funciones de SQL. No obstante, algunas, como las de agrupamiento (AVG, MIN, MAX, COUNT, SUM, STDDEV, etcétera), solamente se pueden usar dentro de cláusulas de selección (SELECT).

Respecto a la utilización de funciones, también debemos tener en cuenta que:

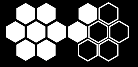
- La función no modifica el valor de las variables o expresiones que se pasan como argumento, sino que devuelve un valor a partir de dicho argumento.

- Si a una función se le pasa un valor nulo en la llamada, normalmente devolverá un valor nulo.

Veamos un ejemplo de funciones: escribiremos un bloque PL/SQL que muestre la fecha y la hora con minutos y segundos.

```
SQL> BEGIN
  2      DBMS_OUTPUT.PUT_LINE(TO_CHAR(SYSDATE,
  3 'DAY " día " DD " de " MONTH " de" YYYY " a las ":HH24
    :MI:SS'));
  4  END;
  5  /
JUEVES   día  02  de  MARZO    de 2006  a las :13:06:10
Procedimiento PL/SQL terminado con éxito.
```





9.8 Comentarios de documentación de los programas

Los comentarios se utilizan para documentar datos generales y particulares de los programas (el autor, la fecha, el objeto del programa, aspectos relevantes o hitos en el programa, función de determinados objetos o comentarios explicativos) e incluso para añadir legibilidad y organización a nuestros programas: líneas de separación, etcétera. En PL/SQL, podemos insertar comentarios en cualquier parte del programa. Estos comentarios pueden ser:

- **De línea con "--".** Todo lo que le sigue en esa línea será considerado comentario. Todos los comentarios incluidos en el código hasta el momento son de este tipo.

- **De varias líneas con "/*" <comentarios> "*/"** (igual que en C). Se pueden incluir en cualquier sección del programa. Es aconsejable, aunque no obligatorio, que incluyan una o varias líneas completas.

```
/* Este programa de prueba ha sido realizado por F. MONTERO
para documentar la realización de comentarios en PL/SQL
*/
  ------------------------------------------------------
  BEGIN   -- aquí comienza la sección ejecutable
```

9.9 Estructuras de control

PL/SQL dispone de estructuras para controlar el flujo de ejecución de los programas.

ESTRUCTURAS ALTERNATIVAS:	
Alternativa simple	
`IF <condicion> THEN instrucciones;END IF;`	Si la condición se cumple, se ejecutan las instrucciones que siguen a la cláusula THEN.
Alternativa doble	
`IF <condicion> THEN instrucciones1;ELSE instrucciones2;END IF;`	Si la condición se cumple, se ejecutan las instrucciones que siguen a la cláusula THEN. En caso contrario, se ejecutarán las instrucciones que siguen a la cláusula ELSE.
Alternativa múltiple con ELSIF	
`IF <condicion1> THEN instrucciones1; ELSIF <condicion2> THEN instrucciones2; ELSIF <condicion3> THEN instrucciones3; ... [ELSE instruccionesotras;] END IF;`	Evalúa, comenzando desde el principio, cada condición, hasta encontrar alguna condición que se cumpla, en cuyo caso ejecutará las instrucciones que siguen a la cláusula THEN correspondiente. La cláusula ELSE es opcional; en caso de que se utilice, se ejecutará cuando no se ha cumplido ninguna de las condiciones anteriores.

La mayoría de las estructuras de control requieren evaluar una condición que en PL/SQL puede dar tres resultados: **TRUE, FALSE** o **NULL.** Pues bien, a efectos de estas estructuras, el valor NULL es equivalente a FALSE, es decir, se considerará que se cumple la condición sólo si el resultado es TRUE. En caso contrario (FALSE o NULL), se considerará que no se cumple.

La estructura CASE no está disponible en versiones anteriores a la 9i. En estos casos deberemos utilizar la alternativa múltiple ELSIF.

Alternativa múltiple con CASE de comprobación	

`CASE expresión`
`WHEN <valorcomprobac1> THEN`
`  instrucciones1;`
`WHEN <valorcomprobac2> THEN`
`  instrucciones2;`
`WHEN <valorcomprobac3> THEN`
`  instrucciones3;`
`...`
`[ELSE`
`  instruccionesotras;]`
`  END CASE;`
	Calcula el resultado de la expresión que sigue a la cláusula CASE. A continuación, comprueba si el valor obtenido coincide con alguno de los valores especificados detrás de las cláusulas WHEN, en cuyo caso ejecutará la instrucción o instrucciones correspondientes. La cláusula ELSE es opcional; se ejecutará en caso de que no se encuentre un valor coincidente en las cláusulas WHEN precedentes.

Alternativa múltiple con CASE de búsqueda	

`CASE`
`WHEN <condicion1> THEN`
`  instrucciones1;`
`WHEN <condicion2> THEN`
`  instrucciones2;`
`WHEN <condicion3> THEN`
`  instrucciones3;`
`...`
`[ELSE`
`      instruccionesotras;]`
`  END CASE;`
	Evalúa, comenzando desde el principio, cada condición, hasta encontrar alguna condición que se cumpla, en cuyo caso ejecutará las instrucciones que siguen a la cláusula THEN correspondiente. La cláusula ELSE es opcional; en caso de que se utilice, se ejecuta cuando no se ha cumplido ninguna de las condiciones anteriores.

Veamos un ejemplo:



Caso práctico

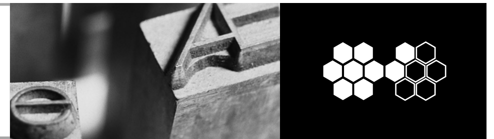
1 Supongamos que pretendemos modificar el salario de un empleado especificado en función del número de empleados que tiene a su cargo:

- Si no tiene ningún empleado a su cargo la subida será 50 €.
- Si tiene 1 empleado la subida será 80 €.
- Si tiene 2 empleados la subida será 100 €.
- Si tiene más de tres empleados la subida será 110 €.

Además, si el empleado es PRESIDENTE se incrementará el salario en 30 €.

```
DECLARE
    v_empleado_no NUMBER(4,0);        -- emple al que subir salario
    v_c_empleados NUMBER(2);          -- cantidad  empl dependen de él
    v_aumento NUMBER(7) DEFAULT 0;    -- importe que vamos a aumentar.
    v_oficio VARCHAR2(10);
```

(Continúa)



(Continuación)

```
BEGIN
    v_empleado_no := &vt_empno;     -- var de sustitución lee nºemple

    SELECT oficio INTO v_oficio FROM emple
        WHERE emp_no = v_empleado_no;

    IF v_oficio = 'PRESIDENTE' THEN -- alternativa simple
        v_aumento := 30;
    END IF;

    SELECT COUNT(*) into v_c_empleados FROM emple
         WHERE dir = v_empleado_no;

    IF v_c_empleados = 0 THEN       -- alternativa múltiple
        v_aumento := v_aumento + 50;
    ELSIF v_c_empleados = 1  THEN
        v_aumento :=  v_aumento + 80;
    ELSIF v_c_empleados = 2 THEN
        v_aumento :=  v_aumento + 100;
    ELSE
        v_aumento :=  v_aumento + 110;
    END IF;

    UPDATE emple SET salario = salario + v_aumento WHERE emp_no = v_empleado_no;
    DBMS_OUTPUT.PUT_LINE(v_aumento);
END;
/
```

El resultado de la ejecución será:

```
Introduzca valor para vt_empno: 7839
140
```

En el programa anterior hemos utilizado una estructura ELSIF pero podíamos haber utilizado una estructura CASE en cualquiera de sus dos formatos:

```
CON CASE DE BÚSQUEDA
--------------------------------
CASE
WHEN v_c_empleados = 0 THEN
    v_aumento := v_aumento + 50;
WHEN v_c_empleados = 1  THEN
    v_aumento :=  v_aumento + 80;
WHEN v_c_empleados = 2 THEN
   v_aumento :=  v_aumento + 100;
ELSE
   v_aumento :=  v_aumento + 110;
END CASE;
```

```
CON CASE DE COMPROBACIÓN
---------------------------------
CASE v_c_empleados
WHEN 0 THEN
   v_aumento := v_aumento + 50;
WHEN 1  THEN
   v_aumento :=  v_aumento + 80;
WHEN 2 THEN
   v_aumento :=  v_aumento + 100;
ELSE
   v_aumento :=  v_aumento + 110;
END CASE;
```




En ocasiones, se puede usar NULL como una instrucción que no hace nada por motivos de claridad o facilidad de codificación:

```
CASE val
WHEN 1 THEN
   INSERT INTO TEMP VALUES 'UNO';
WHEN 2 THEN
   INSERT INTO TEMP VALUES 'DOS';
WHEN 0 THEN
   NULL;                          -- No hace nada.
ELSE THEN
  INSERT INTO TEMP VALUES 'ERROR'
END CASE;
```

ESTRUCTURAS REPETITIVAS

Iterar

```
LOOP
  Instrucciones;
 EXIT [WHEN <condición>];
  instrucciones;
END LOOP;
```

Se trata de un bucle que se repetirá indefinidamente hasta que encuentre una instrucción EXIT sin condición o hasta que se cumpla la condición asociada a la cláusula EXIT WHEN. Es una **condición de salida.**

Mientras

```
WHILE <condicion> LOOP
  instrucciones;
END LOOP;
```

Se evalúa la condición y, si se cumple, se ejecutarán las instrucciones del bucle. El bucle se seguirá ejecutando mientras se cumpla la condición. Es una **condición de continuación.**

En un **bucle WHILE,** si la condición no se cumple al comienzo, no se ejecutará ni una sola línea pues la comprobación se hace antes de entrar en el bucle y posteriormente, una vez finalizadas todas las instrucciones que incluye. **LOOP,** sin embargo, siempre entrará en el bucle, ejecutará las primeras instrucciones y saldrá del bucle cuando se cumpla la condición.

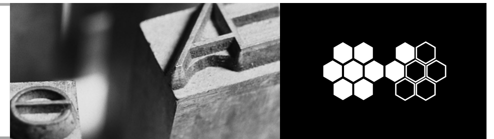


Caso práctico

2 **Supongamos que deseamos analizar una cadena que contiene los dos apellidos para guardar el primer apellido en una variable a la que llamaremos v_1apel. Entendemos que el primer apellido termina cuando encontramos cualquier carácter distinto de los alfabéticos (en mayúsculas).**

```
DECLARE
    v_apellidos VARCHAR2(25);
    v_1apel VARCHAR2(25);
    v_caracter CHAR;
    v_posicion INTEGER :=1;
```

(Continúa)



(Continuación)

```
BEGIN
    v_apellidos := '&vs_apellidos';

    v_caracter := SUBSTR(v_apellidos,v_posicion,1);
    WHILE  v_caracter BETWEEN 'A' AND 'Z' LOOP
        v_1apel := v_1apel || v_caracter;
        v_posicion := v_posicion + 1;
        v_caracter := SUBSTR(v_apellidos,v_posicion,1);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('1er Apellido:'||v_1apel||'*');
END;
/
```

El resultado de la ejecución será:

```
Introduzca valor para vs_apellidos: GIL MORENO
1er Apellido:GIL*
Procedimiento PL/SQL terminado con éxito.
```

El mismo ejemplo con un bucle LOOP… END LOOP será:

```
DECLARE
    v_apellidos VARCHAR2(25);
    v_1apel VARCHAR2(25);
    v_caracter CHAR;
    v_posicion INTEGER :=1;
BEGIN
    v_apellidos := '&vs_apellidos';

    -- desaparece la asignación de v_caracter antes del bucle
    -- se asignará dentro al comienzo del bucle.
    LOOP
        v_caracter := SUBSTR(v_apellidos,v_posicion,1);
      EXIT WHEN v_caracter NOT BETWEEN 'A' AND 'Z';
        v_1apel := v_1apel || v_caracter;
        v_posicion := v_posicion + 1;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('1er Apellido:'||v_1apel||'*');
END;
/
```

EXIT es una instrucción en sí misma (por eso lleva punto y coma al final) y puede ser utilizada con o sin la cláusula WHEN.

```
LOOP
      instrucciones;
      IF <condición> THEN
```

```
            EXIT;
        END IF;
        instrucciones;
    END LOOP;
```

PL/SQL permite la utilización de EXIT incluso en otros bucles y estructuras. Esta práctica es totalmente desaconsejable.

Para
FOR *<variablecontrol>* IN *<valorInicio>*..*<valorFinal>* LOOP   instrucciones; END LOOP;

Donde <variablecontrol> es la variable de control del bucle que se declara de manera implícita como variable local al bucle de tipo BINARY_INTEGER. Esta variable tomará, en primer lugar, el valor especificado en la expresión numérica <valorInicio>, incrementándose en uno para cada nueva iteración hasta alcanzar el valor especificado en la expresión numérica <valorFinal>.

Se utiliza esta estructura cuando se conoce o se puede conocer a priori el número de veces que se debe ejecutar un bucle.

El incremento siempre es una unidad, pero puede ser negativo utilizando la **opción REVERSE:**

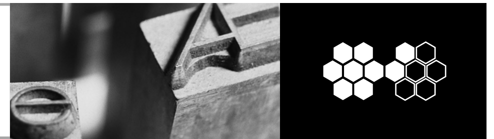
```
FOR <variable> IN REVERSE <valorFinal>..<valorInicio>
   LOOP
     instrucciones;
     ...;
END LOOP;
```

En este caso, comenzará por el valor especificado en segundo lugar e irá restando una unidad en cada iteración:

```
SQL> BEGIN
  2    FOR i IN REVERSE 1..3 LOOP
  3        DBMS_OUTPUT.PUT_LINE(i);
  4    END LOOP;
  5  END;
  6  /

3
2
1
Procedimiento PL/SQL terminado con éxito.
```

Cuando empleamos la opción REVERSE, comenzará por el segundo valor hasta tomar el que sea igual o menor que el especificado en primer lugar. El bloque que se muestra a continuación tiene un error de diseño, ya que no llega siquiera a entrar en el bucle:



```
SQL> BEGIN
  2    FOR i IN REVERSE 5..1 LOOP   -- ERROR
  3        DBMS_OUTPUT.PUT_LINE(i);
  4    END LOOP;
  5  END;
  6  /

Procedimiento PL/SQL terminado con éxito.
```

Podemos indicar los valores mínimo y máximo mediante expresiones:

> PL/SQL no dispone de la **opción STEP** que tienen otros lenguajes, la cual permite especificar incrementos distintos de uno.

```
DECLARE
  Num1 INTEGER;
  Num2 INTEGER;
  ...
BEGIN
  ...
  FOR x IN Num1..Num2 LOOP
    ...
  END LOOP;
  ...
END;
```

Respecto a la variable de control, hay que tener en cuenta que:

- No hay que declararla.
- Es local al bucle y no es accesible desde el exterior del bucle, ni siquiera en el mismo bloque.
- Se puede usar dentro del bucle en una expresión, pero no se le pueden asignar valores.

En el siguiente ejemplo, definimos una variable e intentamos usarla como variable de control. Aun en ese caso la estructura creará la suya propia como local, quedando la nuestra como global en el bucle:

```
<<ppal>>
DECLARE
    i INTEGER;
BEGIN
    ...
    FOR i IN 1..10 LOOP
        ...
/* cualquier referencia a i será entendida como a la
   variable local al bucle. Si quisiéramos referirnos  a
   la otra lo debemos hacer como ppal.i */
    END LOOP;
    ...
    /*La variable local del bucle ya no existe aquí */
END ppal;
```

Veamos algunos ejemplos de aplicación:



Caso práctico

3 **Vamos a construir de dos maneras un bloque PL/SQL que escriba la cadena 'HOLA' al revés.**

Utilizando un bucle FOR

```
SQL> DECLARE
  2    r_cadena VARCHAR2(10);
  3  BEGIN
  4    FOR i IN REVERSE 1..LENGTH('HOLA') LOOP
  5      r_cadena := r_cadena||SUBSTR('HOLA',i,1);
  6    END LOOP;
  7    DBMS_OUTPUT.PUT_LINE(r_cadena);
  8* END;
SQL> /
ALOH
Procedimiento PL/SQL terminado con éxito.
```

Utilizando un bucle WHILE

```
SQL> DECLARE
2    r_cadena VARCHAR2(10);
3    i BINARY_INTEGER;
4  BEGIN
5    i := LENGTH('HOLA');
6    WHILE i >= 1 LOOP
7      r_cadena=r_cadena||SUBSTR('HOLA',i,1);
8       i := i - 1;
9    END LOOP;
10    DBMS_OUTPUT.PUT_LINE(r_cadena);
11* END;
SQL> /
ALOH
Procedimiento PL/SQL terminado con éxito.
```



Actividades propuestas

2 **Escribe un bloque PL/SQL que realice la misma función del ejemplo anterior pero usando un bucle ITERAR.**

A. Utilización de etiquetas

Las **etiquetas** sirven para marcar o nombrar determinadas partes del código del programa como bloques, estructuras de control y otras.

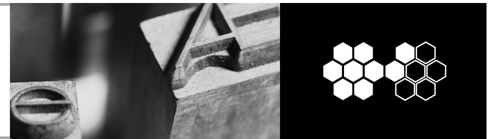
Podemos situar etiquetas en nuestros programas utilizando el formato:

```
<<nombreetiqueta>>
```

Donde los delimitadores `<< >>` forman parte de la sintaxis y `nombretiqueta` representa un identificador válido PL/SQL que utilizaremos después (en este caso, sin los delimitadores) para hacer referencia al elemento.

Podemos etiquetar bucles y otras estructuras para conseguir mayor legibilidad:

```
<<mibucle>>
LOOP
    instrucciones;
    ...
END LOOP mibucle;
```



También se pueden etiquetar las estructuras para eliminar ambigüedades, hacer visibles variables globales y conseguir otras funcionalidades:

```
<<bucleexterno>>
LOOP
    ...
    LOOP
        ...
        EXIT bucleexterno WHEN ... -- sale de ambos bucles
    END LOOP;
END LOOP bucleexterno;
```

En PL/SQL se puede usar la instrucción **GOTO etiqueta.** Para poder utilizar esta orden se deben cumplir las siguientes condiciones:

- No puede haber otra etiqueta en el entorno actual con el mismo nombre.
- La etiqueta debe preceder a un bloque o a un conjunto de órdenes ejecutables.
- La etiqueta no puede estar dentro de un IF, de un LOOP ni de un SUB-BLOQUE internos al bloque donde se produce la llamada.
- Desde una excepción no se puede pasar el control del programa a una etiqueta que está en otra sección del mismo bloque.

Ejemplos de usos correctos

```
BEGIN
   ...
   GOTO insertar_fila;
   ...
   <<insertar_fila>>
   INSERT INTO empleados VALUES ...
END;

También es posible:

BEGIN
   ...
   <<insertar_fila>>
   BEGIN
      INSERT INTO empleados VALUES ...
      ...
   END;
   ...
   GOTO insertar_fila;
   ...
END;
```

Ejemplos de usos ilegales

```
BEGIN
  ...
  FOR i IN 1..10 LOOP
       ...
       GOTO fin_loop;
       ...
       <<fin_loop>> --ilegal, no hay instrucciones ejecutables
   END LOOP;
   ...
   GOTO insertar_fila;
   IF ... THEN
       ...
       <<insertar_fila>>
       INSERT INTO empleados VALUES ...--ilegal, está en un if
       ...
   END IF;
   ...
   GOTO otro_sub;    -- ilegal, está en otro sub-bloque
   ...
   BEGIN
  ...
        <<otro_sub>>
       DELETE FROM ...
   END;
   ...
   <<mi_etiqueta>>
   ...
EXCEPTION
   WHEN ... THEN
       GOTO mi_etiqueta;   --ilegal está en el bloque actual.
END;
...
```

9.10 Subprogramas: procedimientos y funciones

Los **subprogramas** son bloques PL/SQL que tienen un nombre y pueden recibir y devolver valores. Normalmente se guardan en la base de datos y podemos ejecutarlos invocándolos desde otros subprogramas o herramientas.

En todo subprograma podemos distinguir:

- **La cabecera o especificación del subprograma,** que contiene:
  - Nombre del subprograma.
  - Los parámetros con sus tipos (opcional).
  - Tipo de valor de retorno (en el caso de las funciones).
- **El cuerpo del subprograma**. Es un bloque PL/SQL que incluye:
  - Declaraciones (opcional).
  - Instrucciones.
  - Manejo de excepciones (opcional).

En PL/SQL podemos distinguir dos tipos de subprogramas: *procedimientos* y *funciones*. Veámoslos:

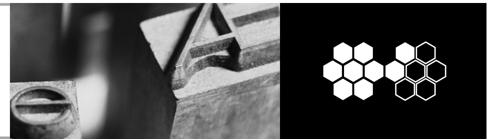
A. Procedimientos

Tienen la siguiente estructura general:

```
PROCEDURE <nombreprocedimiento>
     [( <lista de parámetros> )]
IS
     [<declaraciones>;]
BEGIN
     <instrucciones>;
[EXCEPTION
      <excepciones>;]
END [<nombreprocedimiento>];
```

Podemos apreciar dos partes:

- **La cabecera o especificación del procedimiento.** Comienza con la palabra PROCEDURE y termina después de la declaración de parámetros.
- **El cuerpo del procedimiento.** Corresponde con un bloque PL/SQL. Comienza a continuación de la palabra `IS` (o `AS`) y termina con la palabra `END`, opcionalmente seguida del nombre del procedimiento. En el formato genérico anterior corresponde a la zona sombreada.



Para crear un procedimiento desde SQL*Plus usaremos el siguiente formato:

```
CREATE [OR REPLACE] PROCEDURE <nombreprocedimiento>
[(listadeparametros)]
AS …
```

Por concordancia gramatical se utilizará `AS` en lugar de `IS` al crear un procedimiento o función con la orden CREATE OR REPLACE. Pero, a efectos del compilador, se puede utilizar cualquiera de las dos.

A continuación, se introducirá el bloque de código PL/SQL sin la palabra DECLARE.

La **opción REPLACE** actúa en el caso de que hubiese un subprograma almacenado con ese nombre, sustituyéndolo por el nuevo.

En la lista de parámetros se encuentra la declaración de cada uno de los parámetros que se utilizan para pasar valores al programa separados por comas:

```
(nombreparam1 TIPOP1, nombreparam2 TIPOP2, nombreparam3
TIPOP3, ...)
```

Caso práctico



4 Crearemos un procedimiento que reciba un número de empleado y una cadena correspondiente a su nuevo oficio. El procedimiento deberá localizar el empleado, modificar el oficio y visualizar los cambios realizados.

```
CREATE OR REPLACE PROCEDURE cambiar_oficio (
    num_empleado NUMBER,     -- En los parámetros ..
    nuevo_oficio VARCHAR2)   -- ..no se especifica tamaño
AS
    v_anterior_oficio emple.oficio%TYPE;
BEGIN
    SELECT oficio INTO v_anterior_oficio FROM emple
      WHERE emp_no = num_empleado;

    UPDATE emple SET oficio = nuevo_oficio
         WHERE emp_no = num_empleado;
    DBMS_OUTPUT.PUT_LINE(num_empleado||'*Oficio Anterior:'||v_anterior_oficio||
                                    '*Oficio Nuevo   :'||nuevo_oficio );
END cambiar_oficio;
/
```

El sistema responderá:

```
Procedimiento creado.
```

Ahora el procedimiento está creado y almacenado en la base de datos. Para ejecutarlo podemos invocar el procedimiento desde cualquier herramienta de Oracle, por ejemplo, desde SQL*Plus:

```
SQL>  EXECUTE CAMBIAR_OFICIO(7902,'DIRECTOR');
7902*Oficio Anterior:ANALISTA*Oficio Nuevo   :DIRECTOR
Procedimiento PL/SQL terminado con éxito.
```



Actividades propuestas

3 **Escribe un procedimiento con funcionalidad similar al ejemplo anterior, que recibirá un número de empleado y un número de departamento y asignará al empleado el departamento indicado en el segundo parámetro.**

Podemos invocar al procedimiento desde otro bloque, procedimiento o función. Para ello escribiremos el nombre del procedimiento seguido de la lista de parámetros entre paréntesis, y de un punto y coma:

```
nombreprocedimientoalquellamamos(listadeparametros);
```

La llamada a un procedimiento es una instrucción por sí misma. Cuando se produce la llamada, el control pasa al procedimiento llamado hasta que finaliza su ejecución y el control retorna a la línea siguiente a la llamada.

Por ejemplo, podemos crear un bloque que reciba el apellido y el oficio nuevo. El programa buscará el número de empleado y utilizará una llamada al procedimiento anterior para cambiar oficio:

No es obligatorio escribir el nombre de subprograma detrás del END, pero es aconsejable por razones de legibilidad.

```
CREATE OR REPLACE PROCEDURE cam_ofi_ (
  v_apellido  VARCHAR,
  nue_oficio VARCHAR2)
IS
  v_n_empleado emple.emp_no%TYPE;
BEGIN
    SELECT emp_no INTO v_n_empleado FROM emple
      WHERE  apellido = v_apellido;
    cambiar_oficio(v_n_empleado, nue_oficio);
END cam_ofi_;
```

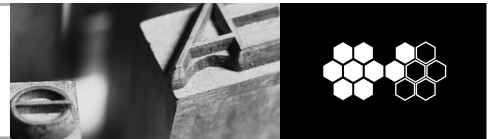
La ejecución del nuevo programa será:

```
SQL> EXECUTE  cam_ofi_('FERNANDEZ','ANALISTA');
7902*Oficio Anterior:DIRECTOR*Oficio Nuevo   :ANALISTA
Procedimiento PL/SQL terminado con éxito.
```

B. Funciones

Las **funciones** tienen una estructura y funcionalidad similar a los procedimientos pero, a diferencia de éstos, las funciones devuelven siempre un valor:

```
FUNCTION <nombredefunción>
      [(<lista de parámetros>)]
RETURN <tipo de valor devuelto >
IS
```



```
      [<declaraciones>;]
BEGIN
      <instrucciones>;
      RETURN <expresión>;
      …
[EXCEPTION
      <excepciones>;]
END [<nombredefunción>];
```

La lista de parámetros es opcional. Si no hay parámetros no debemos poner paréntesis pues dará error igual que en los procedimientos. Sin embargo, es obligatorio el uso de la cláusula **RETURN** en la cabecera y el comando **RETURN** en el cuerpo del programa. No debemos confundirlas:

- La cláusula **RETURN** de la cabecera especifica el tipo del valor que retorna la función.

- En el cuerpo del programa, el comando **RETURN** devuelve el control al programa que llamó a la función, asignando el valor de la expresión que sigue al RETURN a la variable que figura en la llamada a la función.

> Antes de ejecutar un subprograma almacenado, Oracle marca un punto de salvaguarda implícito, de forma que si el subprograma falla durante la ejecución, se desharán todos los cambios realizados por él.

De manera análoga a los procedimientos, para crear o modificar una función utilizaremos el comando **CREATE OR REPLACE FUNCTION:**

> Un procedimiento también puede usar la cláusula RETURN (en este caso, sin devolver ningún valor) para devolver el control al programa que lo llamó, pero no es una técnica recomendable.

```
CREATE OR REPLACE FUNCTION Encontrar_Num_empleado (
    v_apellido  VARCHAR2)
    RETURN REAL
AS
    N_empleado emple.emp_no%TYPE;
BEGIN
    SELECT emp_no INTO N_empleado FROM emple
      WHERE  apellido = v_apellido;
    RETURN N_empleado;
END Encontrar_num_empleado;
```

El formato de llamada a una función consiste en utilizarla como parte de una expresión:

```
<variable> :=  <nombredefunción>[(listadeparámetros)];
```

Para invocar a una función desde SQL también tenemos que «hacer algo» con el valor que devuelve, por ejemplo, utilizarlo como parámetro para otra llamada:

```
SQL> BEGIN DBMS_OUTPUT.PUT_LINE(ENCONTRAR_NUM_EMPLEADO
                                ('GIL'));END;
2  /
7788
Procedimiento PL/SQL terminado con éxito.
```

Una función puede tener varios RETURN pero solo se ejecutará uno de ellos en cada llamada a la función:

```
...
IF nota < 5 THEN
  RETURN 'SUSPENSO';
ELSE
  RETURN 'APROBADO';
END IF;
...
```

C. Parámetros

Los subprogramas utilizan parámetros para pasar y recibir información. Hay dos clases:

- **Parámetros actuales o reales.** Son las variables o expresiones indicadas en la llamada a un subprograma.

- **Parámetros formales.** Son variables declaradas en la especificación del subprograma.

Las declaraciones de variables locales se hacen después del IS, que equivale al DECLARE. En este caso, sí se deberá indicar la longitud pues ya no se trata de parámetros.

Si es necesario, PL/SQL hará la conversión automática de tipos; sin embargo, los tipos de los parámetros actuales y los correspondientes parámetros formales deben ser compatibles.

Podemos hacer el paso de parámetros utilizando la *notación posicional, nominal* o *mixta* (ambas):

- **Notación posicional:** El compilador asocia los parámetros actuales a los formales basándose en su posición.

- **Notación nominal:** El símbolo => después del parámetro actual y antes del nombre del formal indica al compilador la correspondencia.

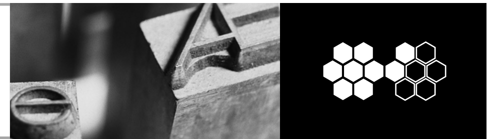
- **Notación mixta:** Consiste en usar ambas notaciones con la restricción de que la notación posicional debe preceder a la nominal.

Por ejemplo, dada la siguiente especificación del procedimiento `ges_dept`:

```
PROCEDURE ges_dept (
  N_departamento INTEGER,
  Localidad VARCHAR2
IS...
```

Desde el siguiente bloque se podrán realizar las llamadas indicadas:

```
DECLARE
  Num_dep INTEGER;
  Local   VARCHAR(14)
BEGIN
...
  -- posicional  ges_dept(Num_dep, local);
  -- nominal     ges_dept(Num_dep   =>   N_departamento,
                 local => localidad);
```



```
  -- nominal    ges_dept(local => localidad, Num_dep =>
                N_departamento);
  -- mixta      ges_dept(Num_dept, Local => localidad);
 ...
 END;
```

Actividades propuestas

4 **Dado el siguiente procedimiento:**

```
PROCEDURE crear_depart (
    v_num_dept depart.dept_no%TYPE,
    v_dnombre depart.dnombre%TYPE DEFAULT 'PROVISIONAL',
    v_loc     depart.loc%TYPE DEFALUT 'PROVISIONAL')
IS
BEGIN
    INSERT INTO depart
      VALUES (v_num_dept, v_dnombre, v_loc);
END crear_depart;
```

Indica cuáles de las siguientes llamadas son correctas y cuáles incorrectas. En el caso de que sean incorrectas, escribe la llamada correcta usando la notación posicional, siempre que sea posible:

```
crear_depart;
crear_depart(50);
crear_depart('COMPRAS');
crear_depart(50,'COMPRAS');
crear_depart('COMPRAS', 50);
crear_depart('COMPRAS', 'VALENCIA');
crear_depart(50,'COMPRAS', 'VALENCIA');
crear_depart('COMPRAS',50, 'VALENCIA');
crear_depart('VALENCIA', 'COMPRAS');
crear_depart('VALENCIA', 50);
```

PL/SQL soporta tres tipos de parámetros:

Tipo	Características y utilización
IN	Son parámetros de **ENTRADA;** se usan para pasar valores al subprograma. Dentro del subprograma el parámetro actúa como una constante, es decir, no se le puede asignar ningún valor. Por tanto, se sitúa siempre a la derecha del operador de asignación. El parámetro actual puede ser una variable, constante, literal o expresión.
OUT	Son parámetros de **SALIDA;** se usan para devolver valores al programa que hizo la llamada. Dentro del subprograma, el parámetro actúa como una variable no inicializada y no puede intervenir en ninguna expresión, salvo para tomar un valor. Se sitúa siempre a la izquierda del operador de asignación. El parámetro actual debe ser una variable.
IN OUT	Son parámetros de **ENTRADA/SALIDA;** permiten pasar un **valor inicial** y devolver un **valor actualizado.** Dentro del subprograma actúa como una variable inicializada. Puede intervenir en otras expresiones y puede tomar nuevos valores. El **parámetro actual** debe ser una variable.

No podemos asignar valores nuevos a los parámetros formales de entrada pues nos encontraremos con el error:

```
PLS-00363: expression 'VParametro' cannot be used as an assignment target.
```

Si queremos cambiar el valor de un parámetro formal de entrada deberemos declararlo de **tipo IN OUT.**

El formato genérico de la declaración de cada uno de los parámetros es:

```
<nombrevariable> [ IN | OUT | IN OUT ] < tipodedato>
[ { := | DEFAULT } <valor>]
```

Debemos tener en cuenta, además, las siguientes reglas:

- Al indicar los parámetros debemos especificar el tipo, pero no el tamaño.
- En el caso de que el subprograma no tenga parámetros no se pondrán los paréntesis.
- Cuando un subprograma recibe un parámetro en modo OUT y se produce una excepción no tratada, el parámetro actual correspondiente queda sin ningún valor.

Valores por defecto en el paso de parámetros de entrada (modo IN): los parámetros de entrada (todos los que hemos manejado hasta este momento) se pueden inicializar con valores por omisión, es decir, indicando al subprograma que en el caso de que no se pase el parámetro correspondiente, asuma un valor por defecto. Para ello, se utiliza la opción DEFAULT <valor>, o bien := <valor>.



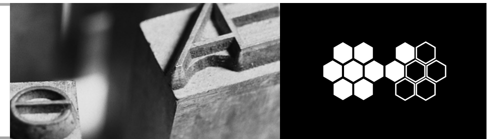
Caso práctico

5 **Supongamos que nos han solicitado un programa de cambio de divisas para un banco que cumpla las siguientes especificaciones:**

- Recibirá una cantidad en euros y el cambio (divisas/euro) de la divisa.
- También podrá recibir una cantidad correspondiente a la comisión que se cobrará por la transacción. En el caso de que no reciba dicha cantidad el programa calculará la comisión que será de un 0,2% del importe, con un mínimo de 3 euros.
- El programa calculará la comisión, la deducirá de la cantidad inicial y calculará el cambio en la moneda deseada, retornando estos dos valores (comisión y cambio) a los parámetros actuales del programa que realice la llamada para solicitar el cambio de divisas.

```
CREATE OR REPLACE PROCEDURE cambiar_divisas (
    cantidad_euros    IN     NUMBER, -- parámetro entrada
    cambio_actual     IN     NUMBER, -- parámetro entrada
    cantidad_comision IN OUT  NUMBER,-- parámetro de e/s
    cantidad_divisas  OUT    NUMBER) -- parámetro de salida
AS
    pct_comision    CONSTANT NUMBER (3,2) := 0.2;
    minimo_comision CONSTANT NUMBER (6) DEFAULT 3;
BEGIN
    IF cantidad_comision IS NULL THEN
         cantidad_comision := GREATEST(cantidad_euros/100*pct_comision,
                                       minimo_comision);
```

(Continúa)



(Continuación)

```
    END IF;
    cantidad_divisas := (cantidad_euros - cantidad_comision) * cambio_actual;
END;
/
```

Una vez creado el procedimiento podremos diseñar programas que hagan uso de él teniendo en cuenta que los parámetros formales para llamar al programa deberán ser cuatro. De éstos, los dos últimos deberán ser variables, que recibirán los valores de la ejecución del programa, tal como aparece en el siguiente procedimiento:

```
CREATE OR REPLACE PROCEDURE mostrar_cambio_divisas (
    eur NUMBER,
    cambio NUMBER)
AS
    v_comision NUMBER (9);
    v_divisas NUMBER (9);
BEGIN
    Cambiar_divisas(eur, cambio, v_comision, v_divisas);
    DBMS_OUTPUT.PUT_LINE ('Euros            : '||
                   TO_CHAR( Eur, '999,999,999.999'));
     DBMS_OUTPUT.PUT_LINE ('Divisas X 1 euro : '||
                   TO_CHAR( cambio, '999,999,999.999'));
     DBMS_OUTPUT.PUT_LINE ('Euros Comisión   : '||
               TO_CHAR( v_comision, '999,999,999.999'));
     DBMS_OUTPUT.PUT_LINE ('Cantidad divisas : '||
               TO_CHAR( v_divisas, '999,999,999.999'));
END;
/
```

Llamamos al programa pasándole la cantidad y el cambio respecto al euro de la divisa queremos cambiar a euros.

```
SQL> EXECUTE MOSTRAR_CAMBIO_DIVISAS(2500, 1.220);
Euros            :       2,500.000
Divisas X 1 euro :           1.220
Euros Comisión   :           5.000
Cantidad divisas :       3,044.000

Procedimiento PL/SQL terminado con éxito.
```

D. Subprogramas almacenados

Los subprogramas (procedimientos y funciones) que hemos visto hasta ahora se pueden compilar independientemente y almacenar en la base de datos Oracle.

Cuando creamos procedimientos o funciones almacenados desde SQL*Plus utilizando los comandos CREATE PROCEDURE o CREATE FUNCTION, Oracle automáticamente compila el código fuente, genera el código objeto (llamado *P-código*) y los guarda en el diccionario de datos. De este modo, quedan disponibles para su utilización.

Oracle dispone de opciones avanzadas y funcionalidades especiales para el paso de parámetros que pueden resultar interesantes para el manejo de estructuras complejas. Estas opciones exceden del objetivo de este libro.

Los programas almacenados tienen dos estados: *disponible* (*valid*) y *no disponible* (*invalid*). Si alguno de los objetos referenciados por el programa ha sido borrado o alterado desde la última compilación del programa, quedará en situación de «no disponible» y se compilará de nuevo automáticamente en la próxima llamada. Al compilar de nuevo, Oracle determina si hay que compilar algún otro subprograma referido por el actual. Se puede producir una cascada de compilaciones.

Estos estados se pueden comprobar en la vista USER_OBJECTS:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS FROM
USER_OBJECTS WHERE OBJECT_NAME='CAMBIAR_OFICIO';

OBJECT_NAME              OBJECT_TYPE             STATUS
------------------       ---------------         -------
CAMBIAR_OFICIO           PROCEDURE               VALID
```

Podemos acceder al código fuente almacenado mediante la vista **USER_SOURCE:**

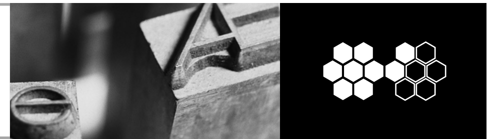
```
SQL> SELECT LINE, SUBSTR(TEXT,1,60) FROM USER_SOURCE WHERE
NAME = 'CAMBIAR_OFICIO';
 LINE  SUBSTR(TEXT,1,60)
------ ---------------------------------------------------
   1   PROCEDURE cambiar_oficio (
   2       num_empleado NUMBER,     -- En los parámetros ..
   3       nuevo_oficio VARCHAR2)   -- ..no se especifica tamaño
   4   AS
   5       v_anterior_oficio emple.oficio%TYPE;
   6   BEGIN
   7       SELECT oficio INTO v_anterior_oficio FROM emple
   8         WHERE emp_no = num_empleado;
   9
  10       UPDATE emple SET oficio = nuevo_oficio
  11            WHERE emp_no = num_empleado;
  12       DBMS_OUTPUT.PUT_LINE(num_empleado||
  13         '*Oficio Anterior:'||v_anterior_oficio||
  14         '*Oficio Nuevo   :'||nuevo_oficio );
  15   END cambiar_oficio;
```

Para volver a compilar un subprograma almacenado en la base de datos se emplea la orden ALTER con la opción COMPILE, indicando PROCEDURE o FUNCTION, según el tipo de subprograma:

```
ALTER {PROCEDURE|FUNCTION} nombresubprograma COMPILE;
```

Para borrar un subprograma, igual que para eliminar otros objetos, se usa la orden DROP seguida del tipo de subprograma (PROCEDURE o FUNCTION):

```
DROP {PROCEDURE|FUNCTION} nombresubprograma;
```



E. Subprogramas locales

Los **subprogramas locales** son procedimientos y funciones que se crean dentro de otro subprograma o de un bloque, al final de la sección declarativa:

```
CREATE OR REPLACE pr_ejem1 /* programa ppal. (contiene el
local)  */
  AS
  ... /* lista de declaraciones: variables, etc. */

  PROGRAM sprloc1 /* comienza el subprograma local */
  ...  /*lista de parámetros del subprograma local */
  IS
  ...  /*declaraciones locales al subprograma local */
  BEGIN
  ...  /*instrucciones del subprograma local */
  END;

BEGIN
  ...
  sprloc1; /* llamada al subprograma local */
  ...
END pr_ejem1;
```

Estos subprogramas tienen las siguientes particularidades:

- Se declaran al final de la sección declarativa de otro subprograma o bloque.
- Sólo están disponibles en el bloque en que se crearon y los bloques hijos de éste. Se les aplica las mismas reglas de ámbito y visibilidad que a las variables declaradas en el mismo bloque.
- Se utilizará este tipo de subprogramas cuando no se contemple su reutilización por otros subprogramas (distintos de aquel en el que se declaran).
- En el caso de subprogramas locales con referencias cruzadas o de subprogramas mutuamente recursivos, hay que realizar *declaraciones anticipadas*, tal como se explica a continuación.

PL/SQL necesita que todos los identificadores, incluidos los subprogramas, estén declarados antes de usarlos. Por eso, la llamada que aparece en el siguiente subprograma es ilegal:

Las **declaraciones anticipadas** permiten definir subprogramas en el orden que queramos, incluso programas mutuamente recursivos.

```
DECLARE
   ...
   PROCEDURE subprograma1 ( ... ) IS
   BEGIN
      ...
      Subprograma2( ... );  -- > ERR. identificador no
declarado
      ...
   END;
```

```
   PROCEDURE subprograma2 ( ... ) IS
   BEGIN
      ...
   END;
   ...
```

Se podía haber invertido el orden de declaración de los procedimientos anteriores, lo cual hubiera resuelto este caso, pero no sirve para procedimientos mutuamente recursivos. El problema se resuelve utilizando declaraciones anticipadas de subprogramas.

```
DECLARE
   PROCEDURE subprograma2 (...);  -- declaración anticipada
       PROCEDURE subprograma1 ( ... ) IS
   BEGIN
      Subprograma2( ... );      -- > ahora no dará error.
      ...
   END;
   PROCEDURE subprograma2 ( ... ) IS
   BEGIN
      ...
   END;
   ...
```

F. Recursividad

No se deben usar algoritmos recursivos en conjunción con cursores FOR LOOP, ya que se puede exceder el número máximo de cursores abiertos. Siempre se pueden sustituir las estructuras recursivas por bucles.

PL/SQL implementa, al igual que la mayoría de los lenguajes de programación, la posibilidad de escribir subprogramas recursivos:

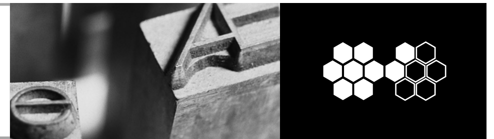
```
CREATE OR REPLACE FUNCTION factorial
  (n POSITIVE)
RETURN INTEGER                          -- > devuelve n!
AS
BEGIN
   IF n = 1 THEN       -- > condición de terminación
      RETURN 1;
   ELSE
      RETURN n * factorial(n - 1); -- > llamada recursiva
   END IF;
END factorial;
```

G. Sobrecarga en los nombres de subprogramas

Teniendo en cuenta las conversiones automáticas y la posibilidad de valores por defecto, la sobrecarga de nombres de programas se debe usar solamente cuando hay total seguridad.

PL/SQL permite sobrecarga en los nombres de subprogramas dentro de paquetes (los veremos más adelante) siempre que los parámetros formales de los subprogramas difieran en número, orden y/o tipo de dato (familia de tipo de dato). Por ejemplo, podemos crear las variables:

- **buscar_emple(NUMBER)** que recibe un número de empleado y lo busca en la base de datos.



- **`buscar_emple(VARCHAR2)`** recibe un nombre, lo busca y muestra los datos encontrados.

En realidad, son dos procedimientos distintos (aunque tengan similitudes, el código es distinto). Oracle los diferencia en las llamadas (y otros comandos) por el contexto, en este caso por los parámetros.

Conceptos básicos



- Los **tipos de datos escalares** estudiados son: *carácter* (CHAR, NCHAR, VARCHAR2, NVARCHAR2…), *numérico* (NUMBER, BINARY_INTEGER, PLS_INTEGER), *booleano*, *Fecha/hora* (DATE, TIMESTAMP…) y otros (ROWID, UROWID,…).

- Los **identificadores** en PL/SQL pueden tener entre 1 y 30 caracteres de longitud; el primer carácter debe ser una letra y los restantes deben ser caracteres alfanuméricos o signos admitidos (letras, dígitos, los signos de dólar, almohadilla y subguión); no pueden incluir signos de puntuación, espacios, etcétera.

- El **formato para declarar una variable** es:

```
<nombre_de_variable> <tipo> [NOT NULL]
[{:= | DEFAULT} <valor>]
```

- Las variables se crean al comienzo del bloque y dejan de existir una vez finalizada la ejecución del bloque en el que han sido declaradas. El ámbito de una variable incluye el bloque en el que se declara y los bloques «hijos» de este.

- Al declarar una constante deberemos incluir CONSTANT y asignar un valor.

- En PL/SQL, podemos insertar comentarios: de línea con -- o de varias líneas con /* … */.

- Las **estructuras de control** son:

  – *Alternativa simple:* IF …THEN …; END IF;

  – *Alternativa doble:* IF…THEN …; ELSE…; END IF;

  – *Alternativa múltiple:* IF… THEN …; ELSIF … THEN …; ELSIF … THEN …; … ELSE …; END IF;

  – *Alternativa multiple:* CASE … WHEN … THEN …; WHEN … THEN …; … ELSE …; END CASE;

  – *Iterar:* LOOP … ; EXIT WHEN … ; END LOOP;

  – *Mientras:* WHILE … LOOP …; END LOOP; -
  Para: FOR I in 1.. n LOOP …; END LOOP;

  – Para crear un *procedimiento* o *función* desde SQL*Plus usaremos CREATE OR REPLACE …

```
PROCEDURE <nombreprocedimiento>
     [( <lista de parámetros> )]
IS   … < BLOQUE PL/SQL > ;

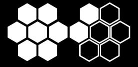
FUNCTION <nombrefunción>
     [( <lista de parámetros> )]
RETURN <tipo d valor devuelto >
IS   … < BLOQUE PL/SQL que incluye
RETURN <expresión>>
```

- La instrucción para invocar un procedimiento es un comando en sí misma. En el caso de las funciones es una expresión que debe hacer algo con el valor devuelto.

- Al indicar los parámetros debemos especificar el tipo pero no el tamaño. En el caso de que el subprograma no tenga parámetros no se pondrán los paréntesis.

- Los parámetros en modo IN, dentro del subprograma, actúan como una constante, es decir, no se les puede asignar ningún otro valor.





Actividades complementarias



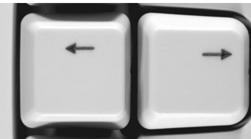
1. Escribe un procedimiento que reciba dos números y visualice su suma.
2. Codifica un procedimiento que reciba una cadena y la visualice al revés.
3. Reescribe el código de los dos ejercicios anteriores para convertirlos en funciones que retornen los valores que mostraban los procedimientos.
4. Escribe una función que reciba una fecha y devuelva el año, en número, correspondiente a esa fecha.
5. Escribe un bloque PL/SQL que haga uso de la función anterior.
6. Desarrolla una función que devuelva el número de años completos que hay entre dos fechas que se pasan como parámetros.
7. Escribe una función que, haciendo uso de la función anterior, devuelva los trienios que hay entre dos fechas (un trienio son tres años).
8. Codifica un procedimiento que reciba una lista de hasta cinco números y visualice su suma.
9. Escribe una función que devuelva solamente caracteres alfabéticos sustituyendo cualquier otro carácter por blancos a partir de una cadena que se pasará en la llamada.
10. Codifica un procedimiento que permita borrar un empleado cuyo número se pasará en la llamada.
11. Escribe un procedimiento que modifique la localidad de un departamento. El procedimiento recibirá como parámetros el número del departamento y la nueva localidad.
12. Visualiza todos los procedimientos y funciones del usuario almacenados en la base de datos y su situación (*valid* o *invalid*).

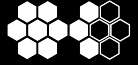
Cursores, excepciones y control de transacciones en PL/SQL

10

En esta unidad aprenderás a:

1. **Utilizar cursores explícitos e implícitos para procesar la información contenida en la base de datos.**
2. **Diseñar programas robustos, capaces de recuperarse ante las condiciones de error que puedan aparecer durante la ejecución, utilizando las técnicas de tratamiento de errores y gestión de excepciones que proporciona PL/SQL.**
3. **Garantizar la integridad de la información utilizando los comandos de control de transacciones.**





10.1 Introducción

En las unidades anteriores hemos estudiado los fundamentos del lenguaje, sus características básicas y aspectos lexicográficos y sintácticos. En esta unidad aprenderemos a manejar el lenguaje PL/SQL en su entorno natural: la gestión segura y eficiente de grandes volúmenes de información de la base de datos.

Para ello, PL/SQL aporta un conjunto de estructuras y comandos que podemos resumir en: la *utilización de cursores*, el *manejo de errores* y el *control de transacciones*. Todos ellos se han comentado someramente en la introducción al lenguaje. En esta unidad profundizaremos en sus características y utilización.

10.2 Cursores

Hasta el momento hemos venido utilizando *cursores implícitos*. Son muy cómodos y sencillos pero plantean diversos problemas. Lo más importante es que la subconsulta debe envolver una fila (y sólo una), de lo contrario, se produciría un error. Por ello, dado que normalmente una consulta devolverá varias filas, se suelen manejar cursores explícitos.

A. Cursores explícitos

Se utilizan para trabajar con consultas que pueden devolver más de una fila.

Hay cuatro operaciones básicas para trabajar con un cursor explícito:

1. **Declaración** del cursor en la zona de declaraciones según el siguiente formato:

   ```
   CURSOR <nombrecursor> IS <sentencia SELECT>;
   ```

2. **Apertura** del cursor en la zona de instrucciones:

   ```
   OPEN <nombrecursor>;
   ```

   La instrucción OPEN ejecuta automáticamente la sentencia SELECT asociada y sus resultados se almacenan en las estructuras internas de memoria manejadas por el cursor.

   No obstante, para acceder a la información debemos dar el paso siguiente.

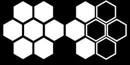
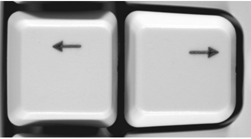
3. **Recogida de información almacenada** en el cursor. Se usa el comando FETCH con el siguiente formato:

   ```
   FETCH <nombrecursor> INTO {<variable>|<listavariables>};
   ```

   Después de INTO figurará:

> Las operaciones permitidas con el cursor son: *declarar, abrir, recoger información* y *cerrar el cursor*. No se le pueden asignar valores ni utilizarlo en expresiones.

> En realidad, un **cursor** es una estructura que apunta a una región de la PGA que contiene toda la información relativa a la consulta asociada, en especial, las filas de datos recuperadas.

– Una variable que recogerá la información de todas las columnas. Puede declararse de esta forma:

```
<variable> <nombrecursor>%ROWTYPE;
```

– O una lista de variables. Cada una recogerá la columna correspondiente de la cláusula SELECT; por tanto, serán del mismo tipo que las columnas.

Cada FETCH recupera una fila y el cursor avanza automáticamente a la fila siguiente en cada nueva instrucción FETCH.

4. **Cierre del cursor.** Cuando el cursor no se va a utilizar hay que cerrarlo:

```
CLOSE <nombrecursor>;
```

El siguiente ejemplo utiliza un cursor explícito para visualizar el nombre y la localidad de todos los departamentos de la tabla DEPART.

```
DECLARE
    CURSOR cur1 IS
       SELECT dnombre, loc FROM depart;      --1.Declarar
    v_nombre VARCHAR2(14);
    v_localidad VARCHAR2(14);
BEGIN
   OPEN cur1;                                --2.Abrir
   LOOP
    FETCH cur1 INTO v_nombre, v_localidad;   --3.Recoger
     EXIT WHEN cur1%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE (v_nombre ||'*'||v_localidad);
   END LOOP;
   CLOSE cur1;                               --4.Cerrar
END;
```

El resultado de la ejecución de este bloque será:

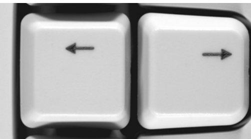
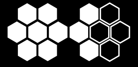
```
CONTABILIDAD*SEVILLA
INVESTIGACION*MADRID
VENTAS*BARCELONA
PRODUCCION*BILBAO
```

Observamos que:

– La sentencia SELECT en la declaración del cursor no contiene la cláusula INTO a diferencia de lo que ocurre en los cursores implícitos.

– La instrucción FETCH se usa para recuperar cada una de las filas seleccionadas por la consulta. Esta información se deposita en las variables que siguen a la cláusula INTO.

– Después de un FETCH debe comprobarse el resultado, lo que se suele hacer preguntando por el valor de alguno de los atributos del cursor que se mencionan en el siguiente apartado.

B. Atributos del cursor

Hay cuatro atributos para consultar detalles de la situación del cursor:

- **%FOUND.** Devuelve verdadero si el último FETCH ha recuperado algún valor; en caso contrario, devuelve falso. Si el cursor no estaba abierto devuelve error, y si estaba abierto pero no se había ejecutado aún ningún FETCH, devuelve NULL. Se suele utilizar como condición de continuación en bucles. En el ejemplo anterior se puede sustituir el bucle y la condición de salida por:

```
...
BEGIN
   OPEN curl;
   FETCH curl INTO v_nombre, v_localidad;
   WHILE curl%FOUND LOOP
      DBMS_OUTPUT.PUT_LINE (v_nombre ||'*'||v_localidad);
      FETCH curl INTO v_nombre, v_localidad;
   END LOOP;
   CLOSE curl;
END
```

- **%NOTFOUND.** Hace lo contrario que el atributo anterior. Se suele utilizar como condición de salida en bucles:

```
...
EXIT WHEN curl%NOTFOUND;
...
```

- **%ROWCOUNT.** Devuelve el número de filas recuperadas hasta el momento por el cursor (número de FETCH realizados satisfactoriamente).

- **%ISOPEN.** Devuelve verdadero si el cursor está abierto.

La siguiente tabla muestra los valores de retorno de los atributos del cursor en diferentes situaciones:

		%FOUND	%ISOPEN	%NOTFOUND	%ROWCOUNT
OPEN	Antes Después	Invalid_cursor NULL	F T	Invalid_cursor NULL	Invalid_cursor 0
PRIMER FETCH	Antes Después	NULL T	T T	NULL F	0 1
SIGUIENTES FETCH	Antes Después	T T	T T	F F	1 ...
ULTIMO FETCH	Antes Después	T F	T T	F T	N N
CLOSE	Antes Después	F Invalid_cursor	T F	T Invalid_cursor	N Invalid_cursor

Tabla 10.1.*Valores de retorno de los atributos del cursor en diferentes situaciones.*



Caso práctico

1 **El siguiente ejemplo ilustra lo que hemos visto hasta ahora respecto a cursores y atributos de cursor: se trata de visualizar los apellidos de los empleados pertenecientes al departamento 20 numerándolos secuencialmente.**

```
DECLARE
   CURSOR c1 IS
   SELECT apellido FROM emple WHERE dept_no=20;
   v_apellido VARCHAR2(10);
BEGIN
   OPEN c1;
   LOOP
      FETCH c1 INTO v_apellido;
      DBMS_OUTPUT.PUT_LINE(TO_CHAR(c1%ROWCOUNT,'99.')
                         ||v_apellido);
      EXIT WHEN c1%NOTFOUND;
   END LOOP;
   CLOSE C1;
END;
```

El resultado de la ejecución será:

```
1.SANCHEZ
2.JIMENEZ
3.GIL
4.ALONSO
5.FERNANDEZ
5.FERNANDEZ
```

En este ejemplo observamos que el último FETCH no devuelve ningún valor, no incrementa el atributo %ROWCOUNT y no sobrescribe el valor de la variable del cursor. Es evidente que el programa está mal diseñado, ya que procesa (visualiza) la información supuestamente recuperada antes de comprobar si se ha recuperado información nueva.

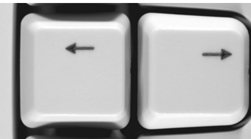
En caso de que FETCH no recupere una nueva fila:

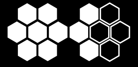
- No se incrementará el atributo %ROWCOUNT.
- No se sobrescribe el valor de la variable del cursor.



Actividades propuestas

1 **Escribe el ejercicio anterior subsanando el error de diseño para que no aparezca el último empleado duplicado. Hacerlo primero manteniendo el bucle LOOP…EXIT WHEN, y posteriormente probar con un bucle WHILE. Observa las diferencias en ambos casos.**





C. Variables de acoplamiento en el manejo de cursores

La cláusula SELECT del cursor deberá seleccionar frecuentemente las filas de acuerdo con una condición. Cuando se trabaja con SQL interactivo se introducen los términos exactos de la condición. Veamos un ejemplo:

```
SQL> SELECT apellido FROM emple
     WHERE dept_no = 20;
```

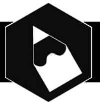
Pero en un programa PL/SQL los términos exactos de esta condición solamente se conocen en tiempo de ejecución. Esta circunstancia obliga a utilizar un diseño más abierto. Las variables de acoplamiento cumplen esta función. Su forma de uso suele ser:

1. Se declara la variable como cualquier otra.

2. Se utiliza la variable en la sentencia SELECT como parte de la expresión.

```
CURSOR nombrecursor IS clausulaselectconvariableacoplam;
```

Caso práctico



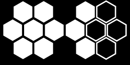
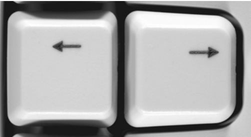
2 **En el siguiente ejemplo se visualizan los empleados de un departamento cualquiera usando variables de acoplamiento:**

```
CREATE OR REPLACE PROCEDURE ver_emple_por_dept (
dep VARCHAR2)
AS
   v_dept NUMBER(2);
   CURSOR c1 IS
       SELECT apellido FROM emple WHERE dept_no = v_dept;
   v_apellido VARCHAR2(10);
BEGIN
   v_dept := dep;
   OPEN c1;
   FETCH c1 INTO v_apellido;
   WHILE c1%FOUND LOOP
       DBMS_OUTPUT.PUT_LINE(v_apellido);
       FETCH c1 INTO v_apellido;
   END LOOP;
  CLOSE c1;
END;
```

El programa sustituirá la variable por su valor en el momento en que se abre el cursor, y se seleccionarán las filas según dicho valor. Aunque ese valor cambie durante la recuperación de los datos con FETCH, el conjunto de filas que contiene el cursor no variará.

También podíamos haber usado directamente el parámetro formal *dep* en lugar de la variable *v_dept*. El resultado es el mismo.

(Continúa)

(Continuación)

Una vez creado el procedimiento, se puede ejecutar:

```
SQL> EXECUTE ver_emple_por_dept(30);
  ARROYO
  SALA
  MARTIN
  NEGRO
  TOVAR
  JIMENO
```



Actividades propuestas

2 **Escribe un procedimiento que reciba una cadena y visualice el apellido y el número de empleado de todos los empleados cuyo apellido contenga la cadena especificada. Al finalizar, visualiza el número de empleados mostrados. El procedimiento empleará variables de acoplamiento para la selección de filas y los atributos del cursor estudiados en el epígrafe anterior.**

D. Cursores FOR…LOOP

Al escribir la sentencia SELECT del cursor debemos cuidar que seleccione únicamente aquellas filas con las que va a trabajar el programa pues de lo contrario sobrecargaremos el sistema y deberemos emplear código adicional para filtrar o tratar las filas requeridas.

Como ya hemos visto, en muchas ocasiones el trabajo con un cursor consiste en:

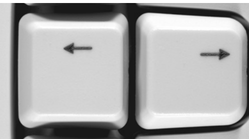
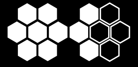
- Declarar el cursor.
- Declarar una variable que recogerá los datos del cursor.
- Abrir el cursor.
- Recuperar con FETCH una a una las filas extraídas, introduciendo los datos en la variable, procesándolos y comprobando también si se han recuperado datos o no.
- Cerrar el cursor.

La estructura de cursores FOR...LOOP simplifica estas tareas realizando todas ellas, excepto la declaración del cursor, de manera implícita.

El formato y el uso de esta estructura es:

1. Se declara el cursor en la sección declarativa (como cualquier otro cursor).

```
CURSOR <nombrecursor> IS <sentencia SELECT>;
```

2. Se procesa el cursor utilizando el siguiente formato:

```
FOR <nombrevareg>  IN <nombrecursor>  LOOP
        ...
END LOOP;
```

Donde *nombrevareg* es el nombre de la variable de registro que creará el bucle para recoger los datos del cursor.

Al entrar en el bucle:

- Se abre el cursor de manera automática.
- Se declara implícitamente la variable *nombrevareg* de tipo *nombrevareg* %ROWTYPE y se ejecuta un FETCH implícito, cuyo resultado quedará en *nombrevareg*.
- A continuación, se realizarán las acciones que correspondan hasta llegar al END LOOP, que sube de nuevo al FOR...LOOP ejecutando el siguiente FETCH impícito, y depositando otra vez el resultado en *nombrevareg,* y así sucesivamente, hasta procesar la última fila de la consulta. En ese momento, se producirá la salida del bucle y se cerrará automáticamente el cursor.

En el siguiente ejemplo, se visualizan el apellido, el oficio y la comisión de los empleados cuya comisión supera 500 € utilizando CURSOR FOR...LOOP:

```
DECLARE
   CURSOR mi_cursor IS
       SELECT apellido, oficio, comision FROM emple
         WHERE comision > 500;
BEGIN
   FOR v_reg IN mi_cursor LOOP
       DBMS_OUTPUT.PUT_LINE(v_reg.apellido||'*'||
         v_reg.oficio||'*'||TO_CHAR(v_reg.comision));
   END LOOP;
END;
```

El resultado será:

```
SALA*VENDEDOR*650
MARTIN*VENDEDOR*1020
```

Cabe subrayar que la variable *nombrevareg* se declara implícitamente y es local al bucle; por tanto, al salir del bucle, la variable de registro no estará disponible.

Dentro del bucle se puede hacer referencia a la variable de registro y a sus campos (cuyo nombre se corresponde con las columnas de la consulta) usando la notación de punto.

El siguiente ejemplo muestra las diferencias en el uso de cursores FOR...LOOP con otras estructuras convencionales.



Caso práctico

3 **Escribiremos un bloque PL/SQL que visualice el apellido y la fecha de alta de todos los empleados ordenados por fecha de alta.**

1. Mediante una estructura cursor FOR…LOOP.

```
DECLARE
   CURSOR c_emple IS
       SELECT apellido, fecha_alt FROM emple
       ORDER BY fecha_alt;
BEGIN
   FOR v_reg_emp IN c_emple LOOP
       DBMS_OUTPUT.PUT_LINE(v_reg_emp.apellido||'*'||
          v_reg_emp.fecha_alt);
   END LOOP;
END;
```

2. Utilizando un bucle WHILE.

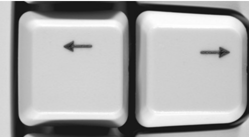
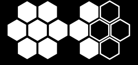
```
DECLARE
   CURSOR c_emple IS
       SELECT apellido, fecha_alt FROM emple
       ORDER BY fecha_alt;
   v_reg_emp c_emple%ROWTYPE;
BEGIN
   OPEN c_emple;
   FETCH c_emple INTO v_reg_emp;
   WHILE c_emple%FOUND LOOP
       DBMS_OUTPUT.PUT_LINE(v_reg_emp.apellido||'*'||
           v_reg_emp.fecha_alt);
       FETCH c_emple INTO v_reg_emp;
   END LOOP;
   CLOSE c_emple;
END;
```

Podemos observar que el bucle FOR…LOOP realiza implícitamente la mayoría de las operaciones con el cursor (abrir, comprobar, recuperar fila y cerrar).



Actividades propuestas

3 **Escribe el procedimiento realizado anteriormente en la actividad 2 pero usando un cursor FOR…LOOP. Observa las diferencias con la estructura anterior. Debemos tener en cuenta que el cursor estará cerrado al salir del bucle y no estarán disponibles sus atributos (en concreto %ROWCOUNT).**

E. Uso de alias en las columnas de selección del cursor

Ya hemos indicado que cuando utilizamos variables de registro declaradas del mismo tipo que el cursor o que la tabla, los campos tienen el mismo nombre que las columnas correspondientes. Cuando esas consultas son expresiones, se puede presentar un problema al tratar de referenciarlas:

```
CURSOR c1 IS
   SELECT dept_no, count(*), sum(salario+NVL(comision,0))
      FROM emple
      GROUP BY dept_no;
```

En estos casos, debemos indicar un alias en la columna:

```
CURSOR c1 IS
SELECT dept_no, count(*) n_emp, sum(salario+NVL(comi-
sion,0)) suma
      FROM emple
      GROUP BY dept_no;
```

F. Cursores con parámetros

En lugar de usar variables de acoplamiento podemos usar parámetros para determinar los términos exactos de la sentencia SELECT cada vez que se abre el cursor.

El uso de parámetros permite que cualquier programa pueda llamar al cursor sin necesidad de conocer y/o manipular las variables de acoplamiento.

La declaración de un cursor con parámetros se realiza indicando la lista de parámetros entre paréntesis a continuación del nombre del cursor. Los parámetros declarados se usarán en la sentencia SELECT de la declaración tal como se muestra a continuación:

```
CURSOR <nombrecursor>  [ (parámetro1, parámetro2, ...)  ]
IS SELECT <sentencia select con parámetros>;
```

Los parámetros formales indicados después del nombre del cursor tienen la siguiente sintaxis:

```
<Nombredeparámetro>  [IN]  <tipodedato>  [{ := | DEFAULT }
<valor>]
```

Los parámetros formales de un cursor son parámetros de entrada. El ámbito de estos parámetros es local al cursor, por eso solamente pueden ser referenciados dentro de la consulta.

```
DECLARE
  ...
  CURSOR cur1
    (v_departamento NUMBER,
     v_oficio VARCHAR2 DEFAULT 'DIRECTOR')
     IS SELECT apellido, salario FROM emple
     WHERE dept_no = v_departamento AND oficio = v_oficio;
```

La apertura del cursor pasándole parámetros se hará:

```
OPEN nombrecursor [ ( parámetro1, parámetro2, ...) ];
```

Donde parámetro1, parámetro2, … son expresiones que contienen los valores que se pasarán al cursor. No tienen por qué ser los mismos nombres de las variables indicadas como parámetros al declarar el cursor; es más, si lo fueran, serían consideradas como variables distintas.

Supongamos, por ejemplo, las siguientes declaraciones:

```
DECLARE
   v_dep emple.dept_no%TYPE;
   v_ofi emple.oficio%TYPE;
   CURSOR cur1
       (v_departamento NUMBER,
        v_oficio VARCHAR2 DEFAULT 'DIRECTOR')
       IS SELECT apellido, salario FROM emple
       WHERE dept_no = v_departamento AND oficio = v_oficio;
 ...
```

El uso de parámetros en cursores es muy habitual en conjunción con paquetes. En la próxima unidad estudiaremos más ejemplos y ejercicios de este tipo.

Cualquiera de los siguientes comandos abrirá el cursor:

```
BEGIN
...
  OPEN cur1(v_dep);
  OPEN cur1(v_dep, v_ofi);
  OPEN cur1(20, 'VENDEDOR');
...
```

Debemos recordar que:

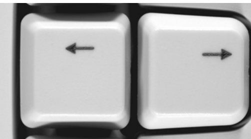
- Los parámetros formales de un cursor son siempre IN y no devuelven ningún valor ni pueden afectar a los parámetros actuales.

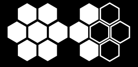
- La recogida de datos se hará, igual que en otros cursores explícitos, con FETCH.

- La cláusula WHERE asociada al cursor se evalúa solamente en el momento de abrir el cursor. En ese momento es cuando se sustituyen las variables por su valor.

En el caso de los cursores FOR…LOOP, puesto que la instrucción OPEN va implícita, el paso de parámetros se hará a continuación del identificador del cursor en la instrucción FOR…LOOP, tal como se muestra en el ejemplo:

```
...
   FOR reg_emple IN cur1(20, 'DIRECTOR') LOOP
...
```





E. Cursores y rupturas de secuencia

Como acabamos de estudiar, el uso de cursores facilita el procesamiento secuencial de los datos recuperados. Esto incluye la posibilidad de rupturas de secuencia según el criterio o criterios determinados por el problema, que deberán ser los criterios de ordenación. A continuación, escribiremos un ejemplo de aplicación de esta técnica.

Caso práctico

4 **Escribe un programa que muestre, en formato similar a las rupturas de control o secuencia vistas en SQL*Plus los siguientes datos:**

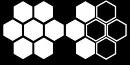
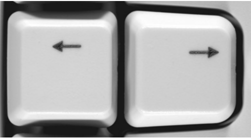
- Para cada empleado: apellido y salario.
- Para cada departamento: número de empleados y suma de los salarios del departamento.
- Al final del listado: número total de empleados y suma de todos los salarios.

```
CREATE OR REPLACE PROCEDURE listar_emple
AS
 CURSOR c1 IS
        SELECT apellido, salario, dept_no FROM emple
ORDER BY dept_no, apellido;
   vr_emp c1%ROWTYPE;
 dep_ant EMPLE.DEPT_NO%TYPE DEFAULT 0;
 cont_emple NUMBER(4) DEFAULT 0;
 sum_sal NUMBER(9,2) DEFAULT 0;
 tot_emple NUMBER(4) DEFAULT 0;
 tot_sal NUMBER(10,2) DEFAULT 0;
BEGIN
 OPEN c1;
 LOOP
      FETCH c1 INTO vr_emp;

/* Si es el primer Fetch inicializamos dep_ant */
      IF c1%ROWCOUNT = 1 THEN
          dep_ant := vr_emp.dept_no;
      END IF;

/* Comprobación nuevo departamento (o finalización) y resumen del anterior e ini-
cialización de contadores y acumuladores parciales  */
IF dep_ant <> vr_emp.dept_no OR c1%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('*** DEPTO: ' || dep_ant ||
            ' NUM. EMPLEADOS: ' || cont_emple ||
                   ' SUM. SALARIOS:  ' || sum_sal);
dep_ant := vr_emp.dept_no;
                        tot_emple := tot_emple + cont_emple;
```

(Continúa)

(Continuación)

```
          tot_sal := tot_sal + sum_sal;
          cont_emple := 0;
          sum_sal := 0;
          END IF;
   EXIT WHEN c1%NOTFOUND; /* Condición de salida del bucle */

/* Escribir Líneas de detalle incrementar y acumular */
     DBMS_OUTPUT.PUT_LINE(RPAD(vr_emp.apellido,10) || ' * '
           ||LPAD(TO_CHAR(vr_emp.salario,'999,999'),12));
       cont_emple := cont_emple + 1;
          sum_sal := sum_sal + vr_emp.salario;

 END LOOP;
 CLOSE c1;

/* Escribir totales informe */
   DBMS_OUTPUT.PUT_LINE(' ****** NUMERO TOTAL EMPLEADOS: '
      || tot_emple || ' TOTAL SALARIOS: ' || tot_sal);

END listar_emple;
```



Actividades propuestas

4 **Haz los cambios necesarios en el programa anterior para que realice el mismo listado usando una estructura de CURSOR FOR…LOOP (hay que tener en cuenta el ámbito de las variables de registro del cursor).**

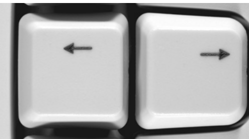
Hecho esto, podemos incluir en el programa rupturas por oficio, indicando en este caso únicamente el nombre del oficio y el número de empleados que tiene. Se entiende que se mantienen las rupturas por departamento y los subtotales; y dentro de cada departamento se harán, rupturas por oficio.

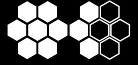
F. Atributos en cursores implícitos

Oracle abre implícitamente un cursor cuando procesa un comando SQL que no esté asociado a un cursor explícito. El cursor implícito se llama **SQL** y dispone también de los cuatro atributos mencionados, que pueden facilitarnos información sobre la ejecución de los comandos SELECT INTO, INSERT, UPDATE y DELETE.

El valor de los atributos del cursor SQL se refiere, en cada momento, a la última orden SQL:

- **SQL%NOTFOUND** dará TRUE si el último INSERT, UPDATE, DELETE o SELECT INTO ha fallado (no ha afectado a ninguna fila).





- **SQL%FOUND** dará TRUE si el último INSERT, UPDATE, DELETE o SELECT INTO ha afectado a una o más filas.

- **SQL%ROWCOUNT** devuelve el número de filas afectadas por el último INSERT, UPDATE, DELETE o SELECT INTO.

- **SQL%ISOPEN** siempre devolverá FALSO, ya que ORACLE cierra automáticamente el cursor después de cada orden SQL.

Estos atributos solamente están disponibles desde PL/SQL, no en órdenes SQL.

Es preciso hacer algunas observaciones respecto al uso de atributos en cursores implícitos pues su comportamiento difiere, en algunos casos, al de los cursores explícitos. De hecho, como acabamos de ver, el cursor estará cerrado inmediatamente después de procesar la orden SELECT...INTO, que es cuando se pregunta por su atributo o atributos (lo cual no determina un error como pasaba en los cursores explícitos).

A continuación, se especifican algunas peculiaridades en el comportamiento y uso de los atributos en cursores implícitos:

- Devolverán un valor relativo a la última orden INSERT, UPDATE, DELETE o SELECT...INTO, aunque el cursor esté cerrado.

- En el caso de que se trate de un SELECT...INTO, debemos tener en cuenta que ha de devolver una fila y sólo una, pues de lo contrario se producirá un error y se levantará automáticamente una excepción:

  - NO_DATA_FOUND, si la consulta no devuelve ninguna fila.

  - TOO_MANY_ROWS, si la consulta devuelve más de una fila.

  Se detendrá la ejecución normal del programa y bifurcará a la sección EXCEPTION. Por tanto, cualquier comprobación de la situación del cursor en esas circunstancias resulta inútil.

- Lo indicado en el párrafo anterior no es aplicable a las órdenes INSERT, UPDATE, DELETE, ya que en estos casos no se levantan las excepciones correspondientes.

El siguiente ejemplo ilustra estas observaciones: se trata de un bloque que pretende cambiar la localidad de un departamento cuyo nombre en este caso es MARKETING (sabiendo que no existe ese departamento).

```
DECLARE
  v_dpto depart.dnombre%TYPE := 'MARKETING'; --(NO existe)
  v_loc  depart.loc%TYPE;
BEGIN
  UPDATE depart SET loc = 'SEVILLA'
      WHERE dnombre = v_dpto;  /* no actualiza ninguna fila
                                pero no levanta excepción */
    IF SQL%NOTFOUND THEN
      DBMS_OUTPUT.PUT_LINE('Error en la actualización');
    END IF;
```

```
    DBMS_OUTPUT.PUT_LINE('Continúa el programa');

    SELECT loc INTO v_loc
       FROM depart
       WHERE dnombre = v_dpto;   /* fallará y levantará
                                    NO_DATA_FOUND */
    IF SQL%NOTFOUND THEN
       DBMS_OUTPUT.PUT_LINE('Nunca pasará por aquí');
    END IF;
END;
```

El resultado de la ejecución del programa será:

```
Error en la actualización
Continúa el programa
...
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line
```

Cuando un SELECT...INTO hace referencia a una función de grupo nunca se levantará la excepción NO_DATA_FOUND, y SQL%FOUND siempre será verdadero. Esto se debe a que las funciones de grupo siempre retornan algún valor (aunque sea NULL o cero).

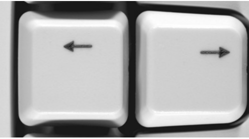
```
BEGIN
    ...
    SELECT MAX(salario) INTO v_max
      FROM emple
      WHERE dept_no = num_depart;  -- > nunca levantará
                                   --  NO_DATA_FOUND
    IF SQL%NOTFOUND THEN           -- > nunca será cierto
        ...                        -- > nunca se ejecutará
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN        -- > no será invocado
      ...
```

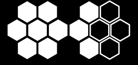
Esta característica resulta útil para comprobar la existencia o no de una determinada fila sin que se levante NO_DATA_FOUND. Por ejemplo, para comprobar si existe un determinado departamento podemos escribir:

```
SELECT COUNT (*) INTO v_dummy
   FROM depart WHERE dept_no = v_depart;
IF v_dummy > 0 THEN ...
```

G. Uso de cursores para actualizar filas

Los cursores se pueden usar para actualizar filas. En este caso, tenemos las siguientes opciones:





- **Cursor FOR UPDATE.** Son cursores que permiten y facilitan la actualización (modificación o eliminación) de las filas seleccionadas por el cursor. Todas las filas seleccionadas serán bloqueadas tan pronto se abra el cursor (OPEN) y serán desbloqueadas al terminar las actualizaciones (al ejecutar COMMIT explícita o implícitamente).

  Para crearlos únicamente habrá que añadirle FOR UPDATE al final de la declaración:

  ```
  CURSOR <nombrecursor> IS <sentencia SELECT del cursor>
  FOR UPDATE;
  ```

  Los cursores así declarados se usan exactamente igual que los cursores explícitos estudiados (OPEN, FETCH, etcétera). Pero además de estas operaciones, permiten actualizar la última fila recuperada con FETCH mediante el comando (UPDATE o DELETE) incluyendo el especificador WHERE CURRENT OF *nombrecursor* en la cláusula para actualizar (UPDATE) o borrar (DELETE).

  El formato para actualizar la fila seleccionada por un cursor FOR UPDATE es:

  ```
  {UPDATE | DELETE} ... WHERE CURRENT OF <nombrecursor>
  ```

  El siguiente procedimiento subirá el salario todos los empleados del departamento indicado en la llamada. La subida será el porcentaje indicado en la llamada:

  ```
  CREATE OR REPLACE PROCEDURE subir_salario_dpto(
      vp_num_dpto NUMBER,
      vp_pct_subida NUMBER)
  AS
      CURSOR c_emple IS SELECT oficio, salario
        FROM emple WHERE dept_no = vp_num_dpto
        FOR UPDATE;
      vc_reg_emple c_emple%ROWTYPE;
      v_inc NUMBER(8,2);
  BEGIN
      OPEN c_emple;
      FETCH c_emple INTO vc_reg_emple;
      WHILE c_emple%FOUND LOOP
          v_inc := (vc_reg_emple.salario / 100) *
            vp_pct_subida;
          UPDATE emple SET salario = salario + v_inc
            WHERE CURRENT OF c_emple; -- (al actual)
          FETCH c_emple INTO vc_reg_emple;
      END LOOP;
  END subir_salario_dpto;
  ```

Si la consulta del cursor hace referencia a múltiples tablas, se deberá usar FOR UPDATE OF `nombrecolumna`, con lo que únicamente se bloquearán las filas correspondientes de la tabla que tenga la columna especificada.

```
CURSOR <nombrecursor> IS <sentencia SELECT del cursor>
FOR UPDATE OF <nombrecolumna>

DECLARE
  ...
  CURSOR c_emple IS SELECT  oficio, salario
    FROM emple, depart
    WHERE emple.dept_no = depart.dept_no
    FOR UPDATE OF salario;
  ...
```

El bloqueo de filas y la determinación de una única transacción de los cursores FOR UPDATE suele ser lo más adecuado cuando queremos actualizar todas las filas seleccionadas por el cursor.

La utilización de la cláusula FOR UPDATE, en ocasiones, puede ser problemática pues:

- Se bloquean todas las filas de la SELECT, no sólo la que se está actualizando en un momento dado.

- Si se ejecuta un COMMIT, después ya no se puede ejecutar FETCH. Es decir, tenemos que esperar a que estén todas las filas actualizadas para confirmar los cambios.

- **Uso de ROWID.** Consiste en usar el identificador de fila (ROWID) como condición de selección para actualizar filas. Para ello procederemos:

  - Al declarar el cursor en la cláusula SELECT, indicaremos que seleccione también el identificador de fila o ROWID:

    ```
    CURSOR nombrecursor IS SELECT col1, col2, ... , ROWID
    FROM tabla;
    ```

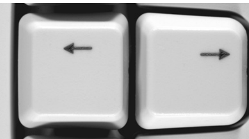
  - Al ejecutar el FETCH, se guardará el número de la fila en una variable o en un campo de la variable del cursor. Después ese número se usará en la cláusula WHERE de la actualización:

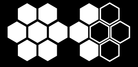
    ```
    {UPDATE | DELETE} ... WHERE ROWID =
    <variable_que_guarda_rowid>
    ```

La actualización de filas con ROWID realiza bloqueos individuales de fila. Normalmente optaremos por ella cuando se prevean pocas actualizaciones en relación con el número de filas seleccionado por el cursor.

En el ejemplo del epígrafe anterior:

```
CREATE OR REPLACE PROCEDURE subir_salario_dpto_b
    (vp_num_dpto NUMBER,
    vp_pct_subida NUMBER)
AS
    CURSOR c_emple IS SELECT oficio, salario, ROWID
      FROM emple WHERE dept_no = vp_num_dpto;
    vc_reg_emple c_emple%ROWTYPE;
    v_inc NUMBER(8,2);
BEGIN
    OPEN c_emple;
    FETCH c_emple INTO vc_reg_emple;
    WHILE c_emple%FOUND LOOP
    v_inc := (vc_reg_emple.salario /100) * vp_pct_subida;
        UPDATE emple SET salario = salario + v_inc
```





```
        WHERE ROWID = vc_reg_emple.ROWID;
      FETCH c_emple INTO vc_reg_emple;
  END LOOP;
END subir_salario_dpto_b;
```

En caso de que tengamos que declarar explícitamente la variable que recogerá el ROWID debemos tener en cuenta que esta pseudocolumna tiene su propio tipo con el mismo nombre (ROWID), tal como estudiamos en la unidad anterior. Por tanto, declararemos la variable así:

```
<nombredevariable> ROWID;
```

Además del uso de cursores FOR UPDATE y del ROWID, podemos seguir usando cualquier condición que permita la selección de las filas a actualizar.

Actividades propuestas



5 **Escribe un programa que incremente el salario de los empleados de un determinado departamento que se pasará como primer parámetro. El incremento será una cantidad en euros que se pasará como segundo parámetro en la llamada. El programa deberá informar del número de filas afectadas por la actualización. Se actualizarán los salarios individualmente y usando el ROWID.**

10.3 Excepciones

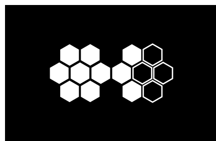
Las **excepciones** sirven para tratar errores en tiempo de ejecución, así como errores y situaciones definidas por el usuario.

Cuando se produce un error PL/SQL levanta una excepción y pasa el control a la sección EXCEPTION, donde buscará un manejador WHEN para la excepción o uno genérico (WHEN OTHERS) y dará por finalizada la ejecución del bloque actual.

Para controlar posibles errores en otros lenguajes que no disponen de gestión de excepciones, se debe controlar después de cada orden cada una de las posibles condiciones de error.

El formato de la sección EXCEPTION es:

```
...
EXCEPTION
    WHEN <NombredeExcepción1> THEN
      <instrucciones1>;
    WHEN <NombredeExcepción2> THEN
      <instrucciones2>;
    ...
    [WHEN OTHERS THEN
       <instrucciones>;]
END <nombre de programa>;
```




A continuación, estudiaremos los tres tipos de excepciones disponibles.

A. Excepciones internas predefinidas

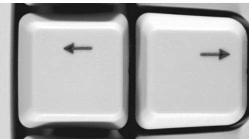
Están predefinidas por Oracle. Se disparan automáticamente al producirse determinados errores. En la tabla adjunta se incluyen las excepciones más frecuentes con los códigos de error correspondientes:

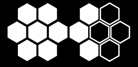
Código error Oracle	Valor de SQL CODE	Excepción	Se disparan cuando...
ORA-06530	-6530	ACCESS_INTO_NULL	Se intenta acceder a los atributos de un objeto no inicializado.
ORA-06531	-6531	COLLECTION_IS_NULL	Se intenta acceder a elementos de una colección que no ha sido inicializada.
ORA-06511	-6511	CURSOR_ALREADY_OPEN	Intentamos abrir un cursor que ya se encuentra abierto.
ORA-00001	-1	DUP_VAL_ON_INDEX	Se intenta almacenar un valor que crearía duplicados en la clave primaria o en una columna con la restricción UNIQUE.
ORA-01001	-1001	INVALID_CURSOR	Se intenta realizar una operación no permitida sobre un cursor (por ejemplo, cerrar un cursor que no estaba abierto).
ORA-01722	-1722	INVALID_NUMBER	Fallo al intentar convertir una cadena a un valor numérico.
ORA-01017	-1017	LOGIN_DENIED	Se intenta conectar a ORACLE con un usuario o una clave no válidos.
ORA-01012	-1012	NOT_LOGGED_ON	Se intenta acceder a la base de datos sin estar conectado a Oracle.
ORA-01403	+100	NO_DATA_FOUND	Una sentencia SELECT ... INTO ... no devuelve ninguna fila.
ORA-06501	-6501	PROGRAM_ERROR	Hay un problema interno en la ejecución del programa.
ORA-06504	-6504	ROWTYPE_MISMATCH	La variable del cursor del HOST y la variable del cursor PL/SQL pertenecen a tipos incompatibles.
ORA-06533	-6533	SUBSCRIPT_OUTSIDE_LIMIT	Se intenta acceder a una tabla anidada o a un *array* con un valor de índice ilegal (por ejemplo, negativo).
ORA-06500	-6500	STORAGE_ERROR	El bloque PL/SQL se ejecuta fuera de memoria (o hay algún otro error de memoria).
ORA-00051	-51	TIMEOUT_ON_RESOURCE	Se excede el tiempo de espera para un recurso.
ORA-01422	-1422	TOO_MANY_ROWS	Una sentencia SELECT ... INTO ... devuelve más de una fila.
ORA-06502	-6502	VALUE_ERROR	Un error de tipo aritmético, de conversión, de truncamiento, etcétera.
ORA-01476	-1476	ZERO_DIVIDE	Se intenta la división entre cero.

Tabla 10.2. *Excepciones más frecuentes en Oracle con los códigos de error.*

No hay que declararlas en la sección DECLARE. Únicamente debemos incluir los manejadores WHEN con el tratamiento para cada excepción y/o un manejador genérico WHEN OTHERS que capturará cualquier otra excepción.

```
DECLARE
    ...
BEGIN
    ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
```





```
    BDMS_OUTPUT.PUT_LINE ('ERROR datos no encontrados');
  WHEN TOO_MANY_ROWS THEN
    BDMS_OUTPUT.PUT_LINE ('ERROR demasiadas filas');
  WHEN OTHERS THEN
    BDMS_OUTPUT.PUT_LINE ('ERROR');
END;
```

B. Excepciones definidas por el usuario

Las excepciones definidas por el usuario se usan para tratar condiciones de error definidas por el programador.

Para su utilización hay que seguir tres pasos:

1. Se declaran en la sección DECLARE de la forma siguiente:

```
<nombreexcepción> EXCEPTION;
```

2. Se disparan o levantan en la sección ejecutable del programa con la orden RAISE:

```
RAISE <nombreexcepción>;
```

3. Se tratan en la sección EXCEPTION según el formato ya conocido:

```
WHEN <nombreexcepción> THEN <tratamiento>;
```

```
DECLARE
    ...
    importe_erroneo EXCEPTION;
    ...
BEGIN
    ...
     IF precio NOT BETWEEN precio_min AND precio_maximo
THEN
        RAISE importe_erroneo;
    END IF;
    ...
EXCEPTION
    ...
    WHEN importe_erroneo THEN
        DBMS_OUTPUT.PUT_LINE('Importe erróneo.
                             Venta cancelada.');
    ...
END;
```

También pueden levantarse excepciones en la sección EXCEPTION, pero no es habitual.

La instrucción RAISE se puede usar varias veces en el mismo bloque con la misma o con distintas excepciones, pero sólo puede haber un manejador WHEN para cada excepción.

```
DECLARE
    ...
    venta_erronea EXCEPTION;
    ...
    importe_erroneo EXCEPTION;
    ...
BEGIN
    ...
    RAISE venta_erronea;
    ...
    RAISE importe_erroneo;
    ...
    RAISE venta_erronea;
    ...
EXCEPTION
    ...
    WHEN importe_erroneo THEN
      ...;
    WHEN venta_erronea THEN
      ...;
END;
```



Caso práctico

5 **El siguiente ejemplo recibe un número de empleado y una cantidad que se incrementará al salario del empleado correspondiente. Utilizaremos dos excepciones, una definida por el usuario `salario_nulo` y la otra predefinida `NO_DATA_FOUND`.**

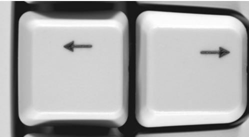
```
CREATE OR REPLACE
PROCEDURE subir_salario(
   num_empleado INTEGER,
   incremento REAL)
IS
   salario_actual REAL;
   salario_nulo EXCEPTION;

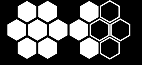
BEGIN
   SELECT salario INTO salario_actual FROM emple
      WHERE emp_no = num_empleado;

   IF salario_actual IS NULL THEN
      RAISE salario_nulo;  -- levanta salario_nulo
   END IF;

   UPDATE emple SET salario = salario + incremento
         WHERE emp_no = num_empleado;
```

(Continúa)





(Continuación)

```
EXCEPTION
   WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE(num_empleado||'*Err.No encontrado');
   WHEN salario_nulo THEN
    DBMS_OUTPUT.PUT_LINE(num_empleado||'*Err. Salario nulo');
END subir_salario;
```

C. Otras excepciones

Existen otros errores internos de Oracle que no tienen asignada una excepción. No obstante, generan (como cualquier otro error de Oracle) un código de error y un mensaje de error, a los que se accede mediante las funciones SQLCODE y SQLERRM.

Cuando se produce uno de estos errores también se transfiere el control a la sección EXCEPTION, donde se tratará el error en la cláusula WHEN OTHERS:

```
EXCEPTION
  ...
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error:'||SQLCODE||SQLERRM);
   ...
END;
```

En este ejemplo, se muestra al usuario el texto 'ERROR' con el *código de error* y el *mensaje de error* utilizando las funciones correspondientes.

Estas dos funciones, SQLCODE y SQLERRM, no son accesibles desde órdenes SQL*Plus, pero se pueden pasar a este entorno a través de soluciones como la siguiente:

```
...
  WHEN OTHERS THEN
    :cod_err := SQLCODE;
    :msg_err := SQLERRM;
    ROLLBACK;
    EXIT;
END;
```

Podemos asociar una excepción de usuario a alguno de estos errores internos que no tienen excepciones predefinidas asociadas. Para ello procederemos así:

1. Definimos la excepción en la sección de declaraciones como si fuese una excepción definida por el usuario:

   ```
   <nombreexcepción> EXCEPTION;
   ```

En realidad, las excepciones predefinidas estudiadas anteriormente no son sino errores de Oracle asociados con excepciones (de manera similar a éstas) en el paquete STANDARD.

2. Asociamos esa excepción a un determinado código de error mediante la directiva del compilador PRAGMA EXCEPTION_INIT, según el formato siguiente:

   PRAGMA EXCEPTION_INIT(<nombre_excepción>, <número_de_error_Oracle>);

3. Indicamos el tratamiento que recibirá la excepción en la sección EXCEPTION como si se tratase de cualquier otra excepción definida por el usuario o predefinida.

```
DECLARE
    ...
    err_externo EXCEPTION;  -- Se define la excepción
    ...
    PRAGMA EXCEPTION_INIT(err_externo, -1547);/* Se asocia
                 con el número de error de Oracle -1547 */
    ...
BEGIN
    ...
    /* no hay que levantar la excepción, pues cuando
       se produzca el error Oracle lo hará */
    ...
EXCEPTION
    ...
    WHEN err_externo THEN    -- Se trata como cualquier otra
          <tratamiento>;
END;
```

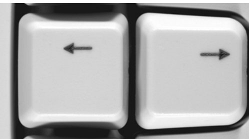


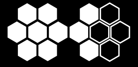
Caso práctico

6 **El siguiente ejemplo ilustra lo estudiado hasta ahora respecto a la gestión de excepciones. Crearemos un bloque donde se define la excepción `err_blancos` asociada con un error definido por el programador y la excepción `no_hay_espacio` asociándola con el error número -1547 de Oracle.**

```
DECLARE
    cod_err number(6);
    vnif varchar2(10);
    vnom varchar2(15);
    err_blancos EXCEPTION;
    no_hay_espacio EXCEPTION;
    PRAGMA EXCEPTION_INIT(no_hay_espacio, -1547);
BEGIN
    SELECT col1, col2 INTO vnif, vnom FROM TEMP2;
     IF SUBSTR(vnom,1,1) <= ' ' THEN
          RAISE err_blancos;
```

(Continúa)





(Continuación)

```
        END IF;
        UPDATE clientes SET nombre = vnom WHERE nif = vnif;
    EXCEPTION
       WHEN err_blancos THEN
           INSERT INTO temp2(col1) VALUES ('ERR blancos');
       WHEN no_hay_espacio THEN
           INSERT INTO temp2(col1) VALUES ('ERR tablespace');
       WHEN NO_DATA_FOUND THEN
           INSERT INTO temp2(col1) VALUES ('ERR no habia datos');
       WHEN TOO_MANY_ROWS THEN
           INSERT INTO temp2(col1) VALUES ('ERdemasiados datos');
       WHEN OTHERS THEN
          cod_err := SQLCODE;
          INSERT INTO temp2(col1) VALUES (cod_err);
    END;
```

Cabe subrayar respecto a las excepciones que aparecen:

- *error_blancos* hay que declararla y levantarla.
- *no_hay_espacio* hay que declararla y asociarla con el error de Oracle, pero no hay que levantarla.
- *NO_DATA_FOUND* y *TOO_MANY_ROWS* no hay que declararlas ni asociarlas. Tampoco hay que levantarlas.
- El manejador WHEN OTHERS cazará cualquier otra excepción e insertará el código de error de Oracle en la tabla *temp2* que suponemos creada.

E. Propagación y ámbito de las excepciones

La gestión de excepciones en PL/SQL tiene unas reglas que se deben considerar en el diseño de aplicaciones:

- Cuando se levanta una excepción en la sección ejecutable, el programa bifurca a la sección EXCEPTION del bloque actual. Si no está definida en ella, la excepción se propaga a la sección EXCEPTION del bloque que llamó al actual; así, hasta encontrar tratamiento para la excepción o devolver el control al programa Host.
- Una vez tratada la excepción en un bloque, se devuelve el control al bloque que llamó al que trató la excepción, con independencia del que la disparó.
- Si, después de tratar una excepción, queremos volver a la línea siguiente a la que se produjo, no podemos hacerlo directamente, pero sí es posible diseñar el programa para que funcione así. Esto se consigue encerrando el comando o comandos que pueden levantar la excepción en un bloque junto el tratamiento para la excepción:

```
SELECT INTO ... -- > Puede levantar NO_DATA_FOUND
...
```

Si queremos que dos o más excepciones ejecuten la misma secuencia de instrucciones, podremos indicarlo en la cláusula WHEN indicando las excepciones unidas por el operador OR:

```
WHEN exc1 OR exc2 OR
exc3… THEN…
```

No obstante, en la lista no podrá aparecer WHEN OTHERS.

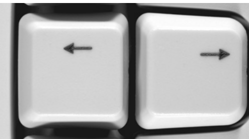
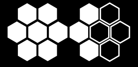
Para que el control del programa no salga del bloque actual cuando se produzca una excepción, podemos encerrar el comando que puede levantar la excepción en un bloque y tratar la excepción en ese bloque:

```
...
BEGIN
  SELECT INTO ...      -- > Puede levantar NO_DATA_FOUND
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    ...;               -- > tratamiento para la excepción
END;
...
```

- La cláusula WHEN OTHERS tratará cualquier excepción que no aparezca en las cláusulas WHEN anteriores, con independencia del tipo de excepción. De este modo, se evita que la excepción se propague a los bloques de nivel superior.

  En el siguiente ejemplo, si se levanta la excepción *ex1*, se tratará en el mismo bloque, pero el control pasará después al bloque `<<exterior>>` en la línea siguiente a la llamada. Si se hubiese levantado NO_DATA_FOUND, al no encontrar tratamiento, la excepción se propaga al bloque `<<exterior>>`, donde será tratada por WHEN OTHERS (suponiendo que no exista un tratamiento específico), devolviendo el control al bloque o la herramienta que llamó a `<<exterior>>`:

```
<<exterior>>
DECLARE
  ...
BEGIN
  ...
    <<interior>>
    DECLARE
        ...
        ex1 EXCEPTION;
    BEGIN
        ...
        RAISE ex1;
        ...
         SELECT col1, col2 INTO ...; --> Levanta
 NO_DATA_FOUND
        ...
    EXCEPTION
        ...
        WHEN ex1 THEN   --> Tratamiento para ex1
            ROLLBACK; --> Después pasará el control
 a (1)
        ...
    END;
  /*(1)*/... ;
  ...
EXCEPTION
  ...
  WHEN OTHERS THEN...
END;
```

- Si la excepción se levanta en la sección declarativa (por un fallo al inicializar una variable, por ejemplo), automáticamente se propagará al bloque que llamó al actual, sin comprobar si existe tratamiento para la excepción en el mismo bloque que se levantó.

> Si la excepción se levanta en la sección declarativa o en la sección EXCEPTION, automáticamente se propagará al bloque que llamó al actual.

- También se puede levantar una excepción en la sección EXCEPTION de forma voluntaria o por un error que se produzca al tratar una excepción. En este caso, también se propagará de forma automática al bloque que llamó al actual, sin comprobar si existe tratamiento para la excepción en el mismo bloque que se levantó. La excepción original (la que se estaba tratando cuando se produjo la nueva excepción) se perderá, ya que solamente puede estar activa una excepción.

```
EXCEPTION
   WHEN INVALID_NUMBER THEN
      INSERT INTO ...-- podría levantar DUP_VAL_ON_INDEX
   WHEN DUP_VAL_ON_INDEX THEN -- no atrapará la exception
```

> Algunos tipos de excepciones se han de tratar con mucha precaución, ya que se puede caer en un bucle infinito (por ejemplo, NOT_LOGGED_ON).

- En ocasiones, puede resultar conveniente volver a levantar la misma excepción después de tratarla para que se propague al bloque de nivel superior. Esto puede hacerse indicando al final de los comandos de manejo de la excepción el comando **RAISE** sin parámetros:

```
...
WHEN TOO_MANY_ROWS THEN
      ...;    -- intrucciones de manejo del error
      RAISE;  -- levanta de nuevo la excepción y la pro-
paga
...
```

- Las excepciones declaradas en un bloque son locales al bloque, no son conocidas en bloques de nivel superior. No se puede declarar dos veces la misma excepción en el mismo bloque, pero sí en distintos bloques. En este caso, la excepción del subbloque prevalecerá sobre la del bloque, aunque esta última se puede levantar desde el subbloque utilizando la notación de punto (RAISE *nombrebloque.nombreexcepción*).

> Si una excepción de usuario no encuentra tratamiento en el bloque en el que ha sido declarada se propagará a bloques de nivel superior; pero éstos sólo pueden cazarlas mediante WHEN OTHERS pues la excepción es desconocida fuera de su ámbito.

- Las variables locales, las globales, los atributos de un cursor, y otros objetos del programa se pueden referenciar desde la sección EXCEPTION según las reglas de ámbito que rigen para los objetos del bloque. Pero si la excepción se ha disparado dentro de un bucle cursor FOR...LOOP, no se podrá acceder a los atributos del cursor, ya que Oracle cierra este cursor antes de disparar la excepción.

- En la sección EXCEPTION no se puede usar < GOTO etiqueta > para salir de esta sección o entrar en otra cláusula WHERE.

- Cuando no se encuentra tratamiento para una excepción en ninguno de los bloques o programas, Oracle da por finalizado con fallos el programa y ejecutará automáticamente un ROLLBACK deshaciendo todos los cambios pendientes de confirmación realizados por el programa.

RAISE_APPLICATION_ERROR tiene un tercer parámetro opcional cuyos detalles de uso quedan fuera de nuestros objetivos.

F. Utilización de RAISE_APPLICATION_ERROR

En el paquete DBMS_STANDARD se incluye un procedimiento muy útil llamado RAISE_APPLICATION_ERROR que sirve para levantar errores y definir y enviar mensajes de error. Su formato es el siguiente:

RAISE_APPLICATION_ERROR(*número_de_error, mensaje_de_error*);

Donde *número_de_error* es un número comprendido entre -20000 y -20999, y *mensaje_de_error* es una cadena de hasta 512 bytes.

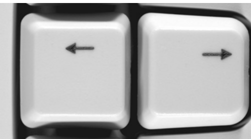
Cuando un subprograma hace esta llamada, se levanta la excepción y produce la salida del programa. Esta excepción solamente puede ser manejada con WHEN OTHERS.

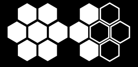


Caso práctico

7 El siguiente ejemplo muestra el funcionamiento de *RAISE_APPLICATION_ ERROR* en un procedimiento de funcionalidad similar al estudiado en el caso práctico 7 (*subir_salario*).

```
CREATE OR REPLACE
PROCEDURE subir_sueldo
   (Num_emple NUMBER, incremento NUMBER)
IS
   salario_actual NUMBER;
BEGIN
   SELECT salario INTO salario_actual FROM empleados
     WHERE emp_no = num_emple;
   IF salario_actual IS NULL THEN
      RAISE_APPLICATION_ERROR(-20010, ' Salario Nulo');
   ELSE
      UPDATE empleados SET sueldo = salario_actual +
        incremento WHERE emp_no = num_emple;
   ENDIF
END subir_sueldo;
```





Actividades propuestas



6 **Escribe un procedimiento que reciba todos los datos de un nuevo empleado y procese la transacción de alta, gestionando posibles errores. El procedimiento deberá gestionar en concreto los siguientes puntos:**

- no_existe_departamento.
- no_existe_director.
- numero_empleado_duplicado.
- Salario nulo: con RAISE_APPLICATION_ERROR.
- Otros posibles errores de Oracle visualizando código de error y el mensaje de error.

10.4 Control de transacciones

Una **transacción** es un conjunto de operaciones dependientes unas de otras que se realizan en la base de datos. Para que la transacción se ejecute han de realizarse todas y cada una de las partes u operaciones que la componen; en el caso de que alguna falle se dará por fallida toda la transacción.

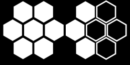
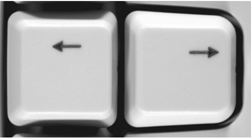
Por ejemplo, en el caso de una transferencia bancaria de una cuenta a otra, la transacción completa ha de contemplar el cargo en una de las cuentas y el abono en la otra. Si por alguna circunstancia una de las dos operaciones (que forman parte de la transacción) no puede llevarse a cabo tampoco debería realizarse la otra.

Una transacción puede incluir una o, frecuentemente, varias instrucciones de manipulación de datos. Será el usuario (el programador, en este caso) quien decide cuántas y cuáles serán las operaciones que compondrán la transacción para garantizar la consistencia de la información almacenada.

En el entorno Oracle la transacción:

- **Comienza** con la primera orden SQL de la sesión del usuario o con la primera orden SQL posterior a la finalización de la transacción anterior.
- **Finaliza** cuando se ejecuta un comando de control de transacciones (COMMIT o ROLLBACK), una orden de definición de datos (DDL) o cuando finaliza la sesión.

Una vez completada la transacción ya no puede deshacerse.

Oracle garantiza la consistencia de los datos en una transacción en términos de VALE TODO o NO VALE NADA, es decir, o se ejecutan todas las operaciones que componen la transacción o no se ejecuta ninguna. Así pues, la base de datos tiene un estado antes de la transacción y un estado después de la transacción, pero no hay estados intermedios.

En el ejemplo de al lado se garantiza que la transferencia se llevará a cabo totalmente o que no se realizará ninguna operación, pero en ningún caso se quedará a medias.

```
BEGIN
   COMMIT; /* marca el final de la transacción anterior
                    y el comienzo de la actual */
   ...
   UPDATE cuentas SET saldo = saldo - v_importe_tranfer
      WHERE num_cta = v_cta_origen;
   UPDATE cuentas SET saldo = saldo + v_importe_tranfer
      WHERE num_cta = v_cta_destino;
  ...
  COMMIT;   /* confirma y da por finalizada la
                   Transacción actual */
  ...
EXCEPTION
   WHEN OTHERS THEN
      ROLLBACK;
END;
```

A. Comandos de control de transacciones

Oracle establece un punto de salvaguarda implícito cada vez que se ejecuta una sentencia de manipulación de datos. En el caso de que la sentencia falle, Oracle restaurará automáticamente los datos a sus valores iniciales.

Oracle dispone de los siguientes comandos de control de transacciones:

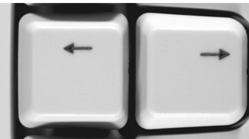
- **COMMIT.** Da por concluida la transacción actual y hace definitivos los cambios efectuados, liberando las filas bloqueadas. Sólo después de que se ejecute el COMMIT los demás usuarios tendrán acceso a los datos modificados.

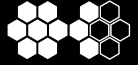
- **ROLLBACK.** Da por concluida la transacción actual y deshace los cambios que se pudiesen haber producido en la misma, liberando las filas bloqueadas. Se utiliza especialmente cuando no se puede concluir una transacción porque se levanta una excepción.

- **ROLLBACK implícitos.** Cuando un subprograma almacenado falla y no se controla la excepción que produjo el fallo, Oracle automáticamente ejecuta ROLLBACK sobre todo lo realizado por el subprograma, salvo que en el subprograma hubiese algún COMMIT, en cuyo caso lo confirmado no sería deshecho.

- **SAVEPOINT** <*punto_de_salvaguarda*>. Se utiliza en conjunción con ROLLBACK TO para poner marcas o puntos de salvaguarda al procesar transacciones. Esto permite deshacer parte de una transacción.

- **ROLLBACK TO** <*punto_de_salvaguarda*>. Deshace el trabajo realizado sobre la base de datos después del punto indicado, incluyendo posibles bloqueos. No obstante, tampoco se confirma el trabajo hecho hasta el *punto_de_salvaguarda*. La transacción no finaliza hasta que se ejecuta un comando de control de transacciones COMMIT o ROLLBACK, o hasta que finaliza la sesión (o se ejecuta una orden de definición de datos DDL).





```
CREATE OR REPLACE PROCEDURE prueba_savepoint (
    numfilas POSITIVE)
AS
BEGIN
  SAVEPOINT NINGUNA;
  INSERT INTO temp1 (col1) VALUES ('PRIMERA FILA');
  SAVEPOINT UNA;
  INSERT INTO temp1 (col1) VALUES ('SEGUNDA FILA');
  SAVEPOINT DOS;
  IF numfilas = 1 THEN
    ROLLBACK TO UNA;
  ELSIF numfilas = 2 THEN
    ROLLBACK TO DOS;
  ELSE
    ROLLBACK TO NINGUNA;
  END IF;
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
```

Los nombres de las marcas son identificadores no declarados y se pueden reutilizar.

Actividades propuestas



7 **Crea el programa anterior y la tabla de pruebas TEMP (Col1 VARCHAR2 (40)). Ejecuta el programa introduciendo diversos valores (0, 1, 2, 3,...), observa y razona su efecto en la tabla.**

El ámbito de los puntos de salvaguarda es el definido por la transacción desde que comienza hasta que termina, por tanto, trasciende de las reglas de ámbito y visibilidad de otros identificadores.

Cuando se ejecuta un ROLLBACK TO <marca>, todas las marcas después del punto indicado desaparecen (la indicada no desaparece). También desaparecen todas las marcas cuando se ejecuta un COMMIT.

B. Transacciones autónomas

Son aquellas que pueden confirmarse o rechazarse con independencia de lo que pueda ocurrir con la otra transacción en curso. Y viceversa, esto es, que lo que ocurra con la transacción en curso no afecte a la transacción autónoma.

Al tratarse de una transacción autónoma, el programa puede ser llamado desde cualquier otro y realizará las acciones previstas sin afectar ni resultar afectado por las transacciones que estén en curso.

Este tipo de transacciones se usan en pequeños programas de un único bloque que realizan tareas auxiliares muy puntuales (por ejemplo, control de acceso de usuarios) que son llamados por distintos programas en diferentes contextos.

Para crear una transacción autónoma crearemos un bloque o programa que realice las acciones previstas e incluiremos en la sección declarativa la directiva o pragma **AUTONOMOUS_TRANSACTION.** Tal como se muestra a continuación:

```
CREATE OR REPLACE
PROCEDURE control_acceso (
    usr VARCHAR2,
    obs VARCHAR2 )
IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO control_accesos
      VALUES (usr, SYSTIMESTAMP, obs);
    COMMIT;
END;
```

C. Transacciones de sólo lectura

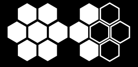
Las transacciones de sólo lectura se usan para garantizar la consistencia de los datos recuperados entre distintas consultas frente a posibles cambios que puedan ocurrir entre ellas.

El comienzo de una transacción de sólo lectura se establece mediante la instrucción SET TRANSACTION READ ONLY. Todas las consultas que se ejecutan a continuación solamente verán aquellos cambios confirmados antes del comienzo de la transacción: es como si se hiciese una fotografía de la base de datos.

Antes de usar SET TRANSACTION READ ONLY debemos confirmar o rechazar la transacción en curso estableciendo así el comienzo de la nueva transacción. Una vez efectuadas todas las operaciones de consulta cuya consistencia queremos garantizar, introduciremos COMMIT para dar por finalizada la transacción de sólo lectura.

```
DECLARE
    ventas_dia  REAL;
    ventas_semana REAL;
BEGIN
  COMMIT; --Confirma transacción anterior e inicia la nueva

  SET TRANSACTION READ ONLY; /* indica que la transacción
                               será de sólo lectura */
  SELECT count(*) INTO ventas_dia FROM ventas
      WHERE fecha  = SYSDATE;
  SELECT count(*) INTO ventas_semana FROM ventas
      WHERE fecha > SYSDATE - 7;
  COMMIT;  -- finaliza la transacción actual de lectura
END;
```

Conceptos básicos



- **Los cursores explícitos** se usan en consultas que pueden devolver más de una fila (o ninguna).

  - Los formatos de las principales operaciones que se realizan con un cursor explícito son:

    Declarar: `CURSOR <nombrecursor> IS SELECT <sentencia select>;`

    Abrir: `OPEN <nombrecursor>;`

    Cerrar: `CLOSE <nombrecursor>;`

    Recuperar fila: `FETCH <nombrecursor> INTO {<variable>|<listavariables>};`

  - Los atributos del cursor sirven para conocer información de su estado. Los principales son: %FOUND, %NOTFOUND, %ISOPEN, %ROWCOUNT

  - **El cursor FOR LOOP** incluye de manera implícita las acciones de abrir, declarar variable para recuperar los datos, recorrer y recuperar las filas y cerrar el cursor. Su formáto genérico es:

    ```
    FOR <nombrevareg> IN <nombrecursor>
    LOOP  <instrucciones;>   END LOOP;
    ```

  - En instrucciones con **cursores implícitos** podemos preguntar por los atributos del cursor aunque estará cerrado.

- **Para actualizar una tabla en PL/SQL** podemos optar por usar en la **cláusula WHERE:**

  - CURRENT OF para indicar que la fila a actualizar es la actual en cursores declarados FOR UPDATE.

  - El ROWID previamente guardado, o cualquier expresión que determine la/s fila/s a actualizar.

- **Cuando se levanta una excepción** el control pasa a la sección EXCEPTION donde buscará un manejador y dará por finalizada la ejecución del bloque actual. Si no encuentra tratamiento, se propagará hasta encontrarlo o retornará al entorno que lanzó la aplicación dando la ejecución de la aplicación por fallida. Una vez tratada la excepción el control retorna a la línea siguiente a la que llamó al programa que trató la excepción.

  - WHEN OTHERS se usa para tratar cualquier excepción que no aparezca en las cláusulas WHEN anteriores.

  - Las excepciones definidas por el usuario deben ser declaradas y levantadas explícitamente según los formatos estudiados.

- **Una transacción** es un conjunto de operaciones dependientes unas de otras. Para que la transacción se complete han de realizarse todas las operaciones que la componen; en el caso de que alguna falle se dará por fallida toda la transacción.

  - La transacción comienza con la primera orden SQL de la sesión o con la primera orden SQL posterior a la finalización de la transacción anterior y finaliza cuando se ejecuta un comando de control de transacciones (COMMIT o ROLLBACK), una orden de definición de datos (DDL) o cuando finaliza la sesión.

  - Cuando un subprograma almacenado falla y no se controla la excepción que produjo el fallo, Oracle automáticamente ejecuta ROLLBACK sobre todo lo realizado por el subprograma, salvo que en el subprograma hubiese algún COMMIT, en cuyo caso lo confirmado no sería deshecho.

  - Para garantizar la consistencia de los datos recuperados entre distintas consultas usaremos SET TRANSACTION READ ONLY antes de ejecutar las instrucciones SELECT; y una vez ejecutadas deberemos incluir COMMIT para liberar la transacción de sólo lectura.

  - Para crear una transacción autónoma crearemos un bloque o programa que realice las acciones previstas e incluiremos en la sección declarativa la directiva o pragma AUTONOMOUS_TRANSACTION. Debemos confirmar o rechazar la transacción antes de salir del ámbito del programa.

Actividades complementarias



1 **Desarrolla un procedimiento que visualice el apellido y la fecha de alta de todos los empleados ordenados por apellido.**

2 **Codifica un procedimiento que muestre el nombre de cada departamento y el número de empleados que tiene.**

3 **Escribe un programa que visualice el apellido y el salario de los cinco empleados que tienen el salario más alto.**

4 **Codifica un programa que visualice los dos empleados que ganan menos de cada oficio.**

5 **Desarrolla un procedimiento que permita insertar nuevos departamentos según las siguientes especificaciones:**

– Se pasará al procedimiento el nombre del departamento y la localidad.

– El procedimiento insertará la fila nueva asignando como número de departamento la decena siguiente al número mayor de la tabla.

– Se incluirá la gestión de posibles errores.

6 **Codifica un procedimiento que reciba como parámetros un número de departamento, un importe y un porcentaje; y que suba el salario a todos los empleados del departamento indicado en la llamada. La subida será el porcentaje o el importe que se indica en la llamada (el que sea más beneficioso para el empleado en cada caso).**

7 **Escribe un procedimiento que suba el sueldo de todos los empleados que ganen menos que el salario medio de su oficio. La subida será del 50 por 100 de la diferencia entre el salario del empleado y la media de su oficio. Se deberá hacer que la transacción no se quede a medias, y se gestionarán los posibles errores.**

8 **Diseña una aplicación que simule un listado de liquidación de los empleados según las siguientes especificaciones:**

– El listado tendrá el siguiente formato para cada empleado:

```
**********************************
Liquidación del empleado       : (1)
Dpto                           : (2)
Oficio                         : (3)
Salario                        : (4)
Trienios                       : (5)
Comp. responsabilidad          : (6)
Comisión                       : (7)
**********************************
Total                          : (8)
**********************************
```

Donde:

– 1, 2, 3 y 4 corresponden a apellido, departamento, oficio y salario del empleado.

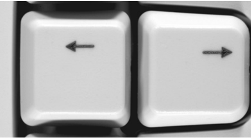
– 5 es el importe en concepto de trienios. Un trienio son tres años completos, desde la fecha de alta hasta la de emisión, y supone 50 €.

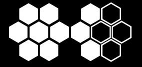
– 6 es el complemento por responsabilidad. Será de 100 € por cada empleado que se encuentre directamente a cargo del empleado en cuestión.

– 7 es la comisión. Los valores nulos serán sustituidos por ceros.

– 8 es la suma de todos los conceptos anteriores.

El listado irá ordenado por Apellido.





9 **Crea la tabla T_liquidacion con las columnas apellido, departamento, oficio, salario, trienios, comp_responsabilidad, comisión y total; y modifica la aplicación anterior para que, en lugar de realizar el listado directamente en pantalla, guarde los datos en la tabla. Se controlarán todas las posibles incidencias que puedan ocurrir durante el proceso.**

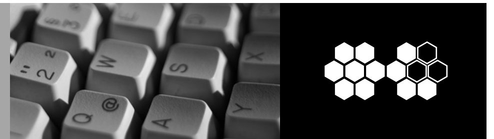
10 **Escribe un programa para introducir nuevos pedidos según las siguientes especificaciones:**

- Recibirá como parámetros PEDIDO_NO, PRODUCTO_NO, CLIENTE_NO, UNIDADES y la FECHA_PEDIDO (opcional, por defecto la del sistema). Verificará todos estos datos así como las unidades disponibles del producto y el límite de crédito del cliente y fallará enviado un mensaje de error en caso de que alguno sea erróneo.
- Insertará el pedido y actualizará la columna DEBE de clientes incrementándola el valor del pedido (UNIDADES * PRECIO_ACTUAL). También actualizará las unidades disponibles del producto e incrementará la comisión para el empleado correspondiente al cliente en un 5% del valor total del pedido. Todas estas operaciones se realizarán como una única transacción.

11 Programación avanzada

En esta unidad aprenderás a:

1. Crear disparadores de base de datos que se ejecuten automáticamente al ocurrir el evento asociado.
2. Definir tipos compuestos según las necesidades y utilizarlos para gestionar la información de la base de datos.
3. Usar paquetes para almacenar los programas y otros objetos utilizados en las aplicaciones aprovechando sus posibilidades adicionales.
4. Manejar los paquetes suministrados por Oracle.
5. Construir programas que permitan crear nuevos objetos de base de datos y modificar las características de los existentes utilizando los paquetes que permiten superar las limitaciones del SQL estático.
6. Definir objetos con métodos asociados, y hacer uso de las facilidades de la Programación Orientada a Objetos en PL/SQL.



11.1 Introducción

En las unidades anteriores hemos utilizado el lenguaje PL/SQL para crear procedimientos que se almacenan de manera aislada en la base de datos y que deben ser invocados explícitamente para que se ejecuten.

En esta unidad aprenderemos a crear disparadores que se ejecutarán automáticamente al producirse determinados eventos, así como a empaquetar los procedimientos junto con variables, tipos, cursores y otros elementos del lenguaje. Definiremos nuestros propios tipos, utilizaremos SQL dinámico para crear y modificar la definición de tablas y usuarios, y trabajaremos con objetos cuyas características se desconocen en tiempo de compilación. También conoceremos las facilidades de la Programación Orientada a Objetos (POO) implementada en PL/SQL.

11.2 Triggers de base de datos

Los ***triggers*** o **disparadores de base de datos** son bloques PL/SQL almacenados que se ejecutan o disparan automáticamente cuando se producen ciertos eventos.

Hay tres tipos de disparadores de bases de datos:

- **Disparadores de tablas.** Asociados a una tabla. Se disparan cuando se produce un determinado suceso o evento de manipulación que afecta a la tabla (inserción, borrado o modificación de filas).

- **Disparaciones de sustitución.** Asociados a vistas. Se disparan cuando se intenta ejecutar un comando de manipulación que afecta a la vista (inserción, borrado o modificación de filas).

- **Disparadores del sistema.** Se disparan cuando ocurre un evento del sistema (arranque o parada de la base de datos, entrada o salida de un usuario, etcétera) o una instrucción de definición de datos (creación, modificación o eliminación de una tabla u otro objeto).

Las operaciones permitidas con el cursor son: ***declarar, abrir, recoger información*** y ***cerrar el cursor***. No se le pueden asignar valores ni utilizarlo en expresiones.

Se suelen utilizar para:

- Implementar restricciones complejas de seguridad o integridad.
- Posibilitar la realización de operaciones de manipulación sobre vistas.
- Prevenir transacciones erróneas.
- Implementar reglas administrativas complejas.
- Generar automáticamente valores derivados.
- Auditar las actualizaciones e, incluso, enviar alertas.

En realidad, un **cursor** es una estructura que apunta a una región de la PGA que contiene toda la información relativa a la consulta asociada, en especial las filas de datos recuperadas.

- Gestionar réplicas remotas de la tabla.
- Etcétera.

A. Creación de un trigger

Aunque existen diferencias dependiendo del tipo de disparador, el formato básico para la creación de un *trigger* es:

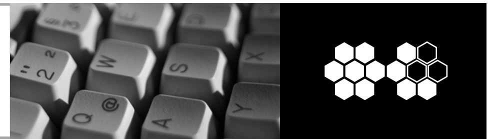
```
CREATE [OR REPLACE]
/* aquí comienza la cabecera del trigger */
TRIGGER nombretrigger
      { BEFORE | AFTER | INSTEAD OF } evento_de disparo
      [ WHEN condición_de_disparo ]
/* aquí comienza el cuerpo del trigger. Es un bloque
PL/SQL */
[DECLARE    ---- opcional
      <declaraciones>]
BEGIN
      <acciones>
[EXCEPTION  ---- opcional
      <gestión de excepciones>]
END];
```

Podemos distinguir:

- **Cabecera del *trigger*.** En ella se define:
  - Nombre del *trigger*: es el nombre o identificador del disparador.
  - Evento de disparo: es el suceso que producirá la ejecución del disparador. Puede ser:
    - Una orden DML (INSERT, DELETE o UPDATE) sobre una tabla o vista.
    - Una orden de definición de datos (CREATE, ALTER, etcétera).
    - O un suceso del sistema.
  - Restricción del *trigger*: condiciona la ejecución del *trigger* al cumplimiento de la condición. Es opcional.
- **Cuerpo del *trigger*.** Es el código que se ejecutará cuando se produzca el evento y cumplan las condiciones especificadas en la cabecera. Se trata de un bloque PL/SQL.

A continuación, estudiaremos cada uno de los tres tipos de disparadores con sus formatos y opciones específicos.

> El cuerpo del *trigger* es un bloque PL/SQL con el formato anteriormente estudiado.
>
> La sección declarativa, si se incluye, comenzará con la cláusula DECLARE.



B. Disparadores de tablas

Son disparadores asociados a una determinada tabla de la base de datos que se disparan cuando se produce un determinado suceso o evento de manipulación que afecta a la tabla (inserción, borrado o modificación de filas).

El siguiente ejemplo crea el *trigger* `audit_subida_salario`, que se disparará después de cada modificación de la columna salario de la tabla EMPLE:

```
CREATE OR REPLACE TRIGGER audit_subida_salario
  AFTER UPDATE OF salario
  ON emple
  FOR EACH ROW
BEGIN
  INSERT INTO auditaremple
     VALUES ('SUBIDA SALARIO EMPLEADO '||:old.emp_no );
END;
 /
Disparador creado.
```

El calificador ***:old*** permite acceder a las columnas de la fila que acaba de ser modificada o borrada.

Tanto ***:old*** como ***:new*** se estudiarán en detalle más adelante.

El disparador insertará una fila en la tabla `auditaremple` con el texto SUBIDA SALARIO EMPLEADO y el número del empleado al que se ha subido el salario.

El formato para la creación de disparadores de tablas es:

```
CREATE [OR REPLACE]
TRIGGER nombretrigger
      { BEFORE | AFTER}
           {DELETE | INSERT | UPDATE [OF <lista_columnas>]}
             ON nombretabla
      [FOR EACH {STATEMENT | ROW    [WHEN (condicion)]}]
< CUERPO DEL TRIGGER (BLOQUE PL/SQL)>
```

Se pueden especificar varios eventos de disparo para un mismo *trigger* utilizando la cláusula OR.

```
 … OR { BEFORE | AFTER}
{DELETE | INSERT | UPDATE
[OF listacolumnas]
```

Cabe señalar que

- El **evento de disparo** será una orden de manipulación: INSERT, DELETE o UPDATE. En el caso de esta última, se podrán especificar opcionalmente las columnas cuya modificación producirá el disparo.
- El **momento en que se ejecuta el *trigger*** puede ser antes (BEFORE) o después (AFTER) de que se ejecute la orden de manipulación.
- El **nivel de disparo del *trigger*** puede ser a *nivel de orden* o a *nivel de fila*.
  - A **nivel de orden** (STATEMENT). El *trigger* se activará una sola vez para cada orden, independientemente del número de filas afectadas por ella. Se puede incluir la cláusula FOR EACH STATEMENT, aunque no es necesario, ya que se asume por omisión.

La combinación de:

- Tres posibles órdenes (INSERT, UPDATE, DELETE).
- Dos posibles momentos de disparo (BEFORE, AFTER).
- Dos niveles de disparo (ROW, STATEMENT).

Da 12 posibles tipos de disparadores:

BEFORE DELETE FOR EACH ROW, AFTER UPDATE, BEFORE INSERT FOR EACH ROW, …

- A **nivel de fila:** el *trigger* se activará una vez para cada fila afectada por la orden. Para ello, se incluirá la cláusula FOR EACH ROW.

- La **restricción del *trigger***. La cláusula WHEN seguida de una condición restringe la ejecución del *trigger* al cumplimiento de la condición especificada. Esta condición tiene algunas limitaciones:

  - Solamente se puede utilizar con *triggers* a nivel de fila (FOR EACH ROW).
  - Se trata de una condición SQL, no PL/SQL.
  - No puede incluir una consulta a la misma o a otras tablas o vistas.

El siguiente ejemplo crea un *trigger* que se disparará cada vez que se borre un empleado, guardando su número de empleado, apellido y departamento en una fila de la tabla `auditaremple`:

```
CREATE OR REPLACE TRIGGER audit_borrado_emple
  BEFORE DELETE
  ON emple
  FOR EACH ROW
BEGIN
  insert INTO auditaremple
    VALUES ('BORRADO EMPLEADO' || '*' ||:old.emp_no ||'*'
            ||:old.apellido || '*Dpto.' ||:old.dept_no);
END;
 /
Disparador creado.
```

Este disparador requiere de la tabla `auditaremple`, que habrá sido creada:

```
CREATE TABLE auditaremple (col1 VARCHAR2(200));
```

Si quisiéramos incluir una restricción para que sólo se ejecute el disparador cuando el empleado borrado sea el PRESIDENTE lo indicaremos insertando una cláusula WHEN antes del cuerpo del *trigger*:

```
WHEN old.oficio = 'PRESIDENTE'
```

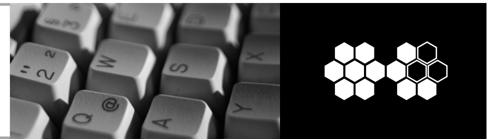
Valores NEW y OLD

Se puede hacer referencia a los valores anterior y posterior a una actualización a nivel de fila. Lo haremos como ***:old.nobrecolumna*** y ***:new.nombrecolumna*** respectivamente.

Por ejemplo: `... IF :new.salario < :old.salario ...`

Al utilizar los valores *old* y *new* deberemos tener en cuenta el evento de disparo:

- Cuando el evento que dispara el *trigger* es DELETE, deberemos hacer referencia a *:old.nombrecolumna*, ya que el valor de *new* es NULL.



- Paralelamente, cuando el evento de disparo es INSERT, deberemos referirnos siempre a ***:****new.nombrecolumna,* puesto que el valor de *old* no existe (es NULL).

- Para los *triggers* cuyo evento de disparo es UPDATE, tienen sentido los dos valores.

Conviene recordar que solamente se puede hacer referencia a los valores ***new*** y ***old*** en disparadores a nivel de fila.

En el caso de que queramos hacer referencia a los valores *new* y *old,* al indicar la restricción del *trigger* (en la cláusula WHEN), lo haremos sin poner los dos puntos. Por ejemplo:

```
WHEN new.salario < old.salario.
```

Si queremos que, en lugar de *old* y *new,* aparezcan otras palabras, podemos usar la cláusula REFERENCING justo antes de WHEN.

Por ejemplo, `...REFERENCING new AS nuevo, old AS anterior ...`

Actividades propuestas



1. **Escribe un disparador que inserte en la tabla `auditaremple(col1 (VARCHAR2(200))` cualquier cambio que supere el 5% del salario del empleado indicando la fecha y hora, el empleado, y el salario anterior y posterior.**

Orden de ejecución de los disparadores

Una misma tabla puede tener varios disparadores. El orden de disparo será el siguiente:

- Antes de comenzar a ejecutar la orden que produce el disparo, se ejecutarán los disparadores BEFORE ... FOR EACH STATEMENT.

- Para cada fila afectada por la orden:

  1. Se ejecutan los disparadores BEFORE ... FOR EACH ROW.

  2. Se ejecuta la actualización de la fila (INSERT UPDATE o DELETE). En este momento se bloquea la fila hasta que la transacción se confirme.

  3. Se ejecutan los disparadores AFTER ... FOR EACH ROW.

- Una vez realizada la actualización se ejecutarán los disparadores AFTER ... FOR EACH STATEMENT.

Observaciones:

- Cuando se dispara un *trigger*, éste forma parte de la operación de actualización que lo disparó, de manera que si el *trigger* falla, Oracle dará por fallida la actualización completa. Aunque el fallo se produzca a nivel de una sola fila, Oracle hará ROLLBACK de toda la actualización.

- En ocasiones, en lugar de asociar varios *triggers* a una tabla, podremos optar por la utilización de un solo *trigger* con múltiples eventos de disparo, tal como se explica en el apartado siguiente.

Múltiples eventos de disparo y predicados condicionales

Un mismo *trigger* puede ser disparado por distintas operaciones o eventos de disparo. Para indicarlo, se utilizará el operador OR.

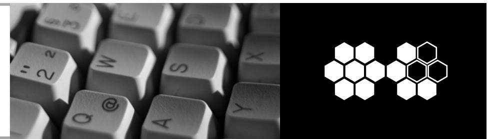
Por ejemplo, `... BEFORE DELETE OR UPDATE ON empleados ...`

En estos casos, es probable que una parte de las acciones del bloque PL/SQL dependa del tipo de evento que disparó el *trigger*. Para facilitar este control Oracle permite la utilización de predicados condicionales que devolverán un valor verdadero o falso para cada una de las posibles operaciones:

INSERTING	Devuelve TRUE si el evento que disparó el *trigger* fue un comando INSERT.
DELETING	Devuelve TRUE si el evento que disparó el *trigger* fue un comando DELETE.
UPDATING	Devuelve TRUE si el evento que disparó el *trigger* fue una instrucción UPDATE.
UPDATING ('nombrecolumna')	Devuelve TRUE si el evento que disparó el *trigger* fue una instrucción UPDATE y la columna especificada ha sido actualizada.

Por ejemplo:

```
CREATE TRIGGER ...
BEFORE INSERT OR UPDATE OR DELETE ON empleados ...
BEGIN
  IF INSERTING THEN
    ...
  ELSIF DELETING THEN
    ...
  ELSIF UPDATING('salario') THEN
    ...
  END IF
  ...
END;
```



Restricciones

El código PL/SQL del cuerpo del *trigger* puede contener instrucciones de consulta y de manipulación de datos, así como llamadas a otros subprogramas. No obstante, existen restricciones que deben ser contempladas:

- El bloque PL/SQL no puede contener sentencias de control de transacciones como COMMIT, ROLLBACK o SAVEPOINT.

- Tampoco se pueden hacer llamadas a procedimientos que transgredan la restricción anterior.

- No se pueden utilizar comandos DDL.

- Un *trigger* no puede contener instrucciones que consulten o modifiquen *tablas mutantes*. Una **tabla mutante** es aquella que está siendo modificada por una sentencia UPDATE, DELETE o INSERT en la misma sesión.

- No se pueden cambiar valores de las columnas que sean claves primaria, única o ajena de *tablas de restricción*. Una **tabla de restricción** es una tabla que debe ser consultada o actualizada directa o indirectamente por el comando que disparó el *trigger* (normalmente, debido a una restricción de integridad referencial) en la misma sesión.

Los *triggers* a nivel de comando (FOR EACH STATEMENT) no se verán afectados por las restricciones que acabamos de enunciar para las tablas mutantes y tablas de restricción, excepto cuando el *trigger* se dispare como resultado de una restricción ON DELETE CASCADE.

C. Disparadores de sustitución

Son disparadores *asociados a vistas* que arrancan al ejecutarse una instrucción de actualización sobre la vista a la que están asociados. Se ejecutan en lugar de (INSTEAD OF) la orden de manipulación que produce el disparo del *trigger;* por eso se denominan disparadores de sustitución.

El formato genérico para la creación de estos disparadores de sustitución es:

```
CREATE [OR REPLACE]
TRIGGER nombretrigger
            INSTEAD OF
          {DELETE | INSERT | UPDATE [OF <lista_columnas>}
           ON nombrevista
     [FOR EACH ROW]     [WHEN (condicion)]
<CUERPO DEL TRIGGER (BLOQUE PL/SQL)>
```

Los disparadores de sustitución se ejecutan siempre a nivel de fila.

La cláusula FOR EACH ROW es opcional pero no hay otra posibilidad.

Los disparadores de sustitución tienen estas características diferenciales:

- Solamente se utilizan en *triggers* asociados a vistas, y son especialmente útiles para realizar operaciones de actualización complejas.
- Actúan siempre a nivel de fila, no a nivel de orden, por tanto, a diferencia de lo que ocurre en los disparadores asociados a una tabla, la opción por omisión es FOR EACH ROW.
- No se puede especificar una restricción de disparo mediante la cláusula WHEN (pero no se puede conseguir una funcionalidad similar utilizando estructuras alternativas dentro del bloque PL/SQL).



Caso práctico

1 Supongamos que disponemos de la siguiente vista:

```
CREATE VIEW EMPLEAD AS
SELECT EMP_NO, APELLIDO, OFICIO, DNOMBRE, LOC
FROM EMPLE, DEPART
WHERE EMPLE.DEPT_NO = DEPART.DEPT_NO;
```

Los usuarios verán los datos:

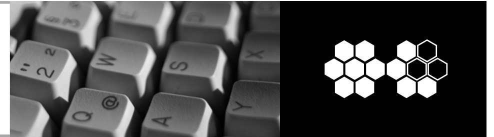
```
SQL> select * from emplead;
    EMP_NO    APELLIDO    OFICIO        DNOMBRE          LOC
    ------    --------    ----------    ---------        -----
    7839      REY         PRESIDENTE    CONTABILIDAD     SEVILLA
    7876      ALONSO      EMPLEADO      INVESTIGACIÓN    MADRID
    7521      SALA        VENDEDOR      VENTAS           BARCELONA
    ...       ...         ...           ...              ...
```

Las siguientes operaciones de manipulación sobre los datos de la vista darán como resultado:

```
SQL> INSERT INTO EMPLEAD VALUES (7999, 'MARTINEZ', 'VENDEDOR', 'CONTABILIDAD', 'SEVILLA');
ERROR en línea 1: ORA-01776: no se puede modificar más de una tabla base a través
de una vista.

SQL> UPDATE EMPLEAD SET DNOMBRE = 'CONTABILIDAD' WHERE APELLIDO = 'SALA';
ERROR en línea 1:ORA-01779: no se puede modificar una columna que se corresponde
 con una tabla reservada por clave
```

(Continúa)



(Continuación)

Para facilitar estas operaciones de manipulación crearemos el siguiente disparador de sustitución:

```
CREATE OR REPLACE TRIGGER t_ges_emplead
INSTEAD OF DELETE OR INSERT OR UPDATE
ON emplead
FOR EACH ROW
DECLARE
    v_dept depart.dept_no%TYPE;
BEGIN
    IF DELETING THEN            /* Si se pretende borrar una fila */
        DELETE FROM EMPLE WHERE emp_no = :old.emp_no;
     ELSIF INSERTING THEN        /* Si se intenta insertar una fila */
        SELECT dept_no INTO v_dept  FROM depart
          WHERE depart.dnombre = :new.dnombre
          AND loc = :new.loc;
        INSERT INTO EMPLE (emp_no, apellido, oficio, dept_no)
          VALUES (:new.emp_no, :new.apellido, :new.oficio, v_dept);
     ELSIF UPDATING('dnombre') THEN      /* Si se trata de actualizar
                                         la columna dnombre*/
        SELECT dept_no INTO v_dept FROM depart
          WHERE dnombre = :new.dnombre;
        UPDATE emple SET dept_no = v_dept
          WHERE emp_no = :old.emp_no;
   ELSIF UPDATING('oficio') THEN      /* Si se pretende actualizar
                                       la columna oficio */
      UPDATE emple SET oficio = :new.oficio
        WHERE emp_no = :old.emp_no;
   ELSE
    RAISE_APPLICATION_ERROR(-20500,'Error en la actualización');
   END IF;
END;
/
```

Ahora podemos realizar las operaciones anteriormente indicadas. Se puede cambiar a un empleado de departamento indicando el nombre del departamento nuevo. El disparador se encargará de comprobar y asignar el número de departamento que corresponda. También se han limitado las columnas que hay que actualizar. En caso de que la operación que pretende el usuario no se contemple entre las alternativas, el *trigger* levantará un error en la aplicación haciendo que falle toda la actualización.

D. Disparadores del sistema

Se disparan cuando ocurre un evento del sistema (arranque o parada de la base de datos, entrada o salida de un usuario, etcétera) o una instrucción de definición de datos (creación, modificación o eliminación de un objeto).

Para crear *triggers* del sistema hay que tener el privilegio ADMINISTER DATABASE TRIGGER.

La sintaxis para su creación es:

```
CREATE [OR REPLACE]
TRIGGER nombretrigger
{BEFORE | AFTER} {<lista eventos definición> | <lista
  eventos del sistema>}
ON {DATABASE | SCHEMA} [WHEN (condicion)]
<CUERPO DEL TRIGGER (BLOQUE PL/SQL)>
```

> Los disparadores STARTUP y SHUTDOWN sólo tienen sentido a nivel ON DATABASE, no asociarlos a un esquema pues, aunque Oracle permite la asociación, nunca se dispararán.

Donde:

– El nombre del *trigger* puede incluir el esquema mediante la notación de punto.

– `<lista eventos definición>` puede incluir uno o más eventos DDL separados por OR.

– `<lista eventos del sistema>` puede incluir uno o más eventos del sistema separados por OR.

– El especificador ON SCHEMA|DATABASE indica el ***nivel de disparo del trigger*:**

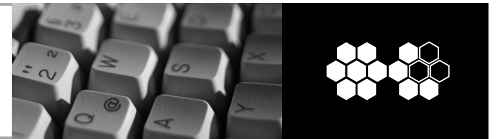
- **ON DATABASE** se disparará siempre que ocurra el evento de disparo.

- **ON SCHEMA** se disparará solamente si el evento ocurre en el esquema determinado por el *trigger*. Por defecto, este esquema es aquel al que pertenece el *trigger,* pero se puede indicar otro mediante la notación de punto: ON esquema.SCHEMA

Al asociar un disparador a un evento del sistema debemos indicar el momento del disparo. Algunos eventos sólo pueden diparar ANTES de producirse, otros DESPUES y otros admiten las dos posibilidades, tal como se muestra en la tabla siguiente:

Evento	Momento	Se disparan:
STARTUP	AFTER	Después de arrancar la instancia.
SHUTDOWN	BEFORE	Antes de apagar la instancia.
LOGON	AFTER	Después de que el usuario se conecte a la base de datos.
LOGOFF	BEFORE	Antes de la desconexión de un usuario.
SERVERERROR	AFTER	Cuando ocurre un error en el servidor.
CREATE	BEFORE | AFTER	Antes o después de crear un objeto en el esquema.
DROP	BEFORE | AFTER	Antes o después de borrar un objeto en el esquema.
ALTER	BEFORE | AFTER	Antes o después de cambiar un objeto en el esquema.
TRUNCATE	BEFORE | AFTER	Antes o después de ejecutar un comando TRUNCATE.
GRANT	BEFORE | AFTER	Antes o después de ejecutar un comando GRANT.
REVOKE	BEFORE | AFTER	Antes o después de ejecutar un comando REVOKE.
DLL	BEFORE | AFTER	Antes o después de ejecutar cualquier comando de definición de datos (excepto algunos, como CREATE DATABASE).
Otros comandos del sistema	BEFORE | AFTER	RENAME, ANALYZE, AUDIT, NO AUDIT, COMMENT, SUSPEND, ASSOCIATE STATISTICS, DISASSOCIATE STATISTICS.

Tabla 11.1. *Eventos del sistema y eventos de definición.*



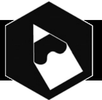
PL/SQL dispone de algunas funciones que permiten acceder a los atributos del evento del disparo ORA_SYSEVENT, ORA_LOGIN_USER, ORA_DICT_OBJ_NAME, ORA_DICT_OBJ_TYPE, etcétera. En los manuales del producto podemos encontrar un listado completo de las funciones accesibles desde estos disparadores. Estas funciones pueden usarse en la cláusula WHEN o en el cuerpo del disparador. Cabe mencionar que:

– Desde un disparador LOGON o LOGOFF se puede comprobar el identificador de usuario, o el nombre de usuario (ID, USERID, USERNAME).

– Desde un disparador DDL puede comprobar el tipo y el nombre del objeto que se está modificando (y el ID o nombre de usuario).

Los STARTUP y SHUTDOWN no pueden llevar la cláusula WHEN pues no permiten condiciones.

Caso práctico



2 Escribiremos un disparador que controlará las conexiones de los usuarios en la base de datos.

Para ello introducirá en la tabla `control_conexiones` el nombre de usuario (USER), la fecha y hora en la que se produce el evento de conexión, y la operación CONEXIÓN que realiza el usuario.

```
CREATE OR REPLACE TRIGGER ctrl_conexiones
AFTER LOGON
ON DATABASE
BEGIN
    INSERT INTO control_conexiones (usuario, momento, evento)
    VALUES (ORA_LOGIN_USER, SYSTIMESTAMP, ORA_SYSEVENT);
END;
```

Para que el disparador pueda crearse deberá estar creada la tabla `control_conexiones`:

```
CREATE TABLE control_conexiones (usuario VARCHAR2(20),
  momento TIMESTAMP, evento VARCHAR2(20));
```

Para crear este disparador a nivel ON DATABASE hay que tener el privilegio ADMINISTER DATABASE TRIGGER, de lo contrario sólo nos permitirá crearlo ON SCHEMA.

Una vez creado el disparador cualquier evento de conexión en el esquema producirá el disparo del *trigger* y la consiguiente inserción de la fila en la tabla.

```
USUARIO        MOMENTO                                 EVENTO
-----------    -----------------------------------     -------------
FERNANDO       29/03/06 10:54:51,145000                LOGON
```

Por otro parte, crearemos un *trigger* que inserte en la tabla `control_eventos` cualquier instrucción de definición de datos:

```
CREATE TABLE control_eventos (usuario VARCHAR2(20), momento TIMESTAMP, evento
VARCHAR2(40));
```

(Continúa)

(Continuación)

```
CREATE OR REPLACE TRIGGER ctrl_eventos
AFTER DDL
ON DATABASE
BEGIN
    INSERT INTO control_eventos (usuario, momento, evento)
    VALUES (USER, SYSTIMESTAMP, ORA_SYSEVENT || '*' ||ORA_DICT_OBJ_NAME);
END;
```

E. Consideraciones y opciones de utilización

Una vez que probemos el disparador deberemos borrarlo con DROP.

- Para crear un *trigger* hay que tener privilegios de CREATE TRIGGER, así como los correspondientes privilegios sobre la tabla o tablas y otros objetos referenciados por el *trigger*.

- Con el nombre del *trigger* se puede especificar el esquema en el que queremos crearlo, utilizando la notación de punto. Por omisión, Oracle asumirá nuestro esquema actual en el momento de crear el *trigger*. El privilegio CREATE ANY TRIGGER permite crear *trigger* en cualquier esquema.

- La tabla, vista u objeto al que se asocia el disparador puede pertenecer a un esquema distinto del actual, siempre que se tengan los privilegios correspondientes; en este caso, se utilizará la notación de punto. No se puede asociar un *trigger* a una tabla del esquema SYS.

- Como ya hemos señalado, un *trigger* forma parte de la operación de actualización que lo disparó y si éste falla, Oracle dará por fallida la actualización. Esta característica puede servir para impedir desde el *trigger* que se realice una determinada operación, ya que si levantamos una excepción y no la tratamos, el *trigger* y la operación fallarán.

- Los disparadores son una herramienta muy útil, pero su uso indiscriminado puede degradar el comportamiento de la base de datos y ser fuente de problemas. Por ello, cuando se trata de implementar restricciones de integridad, se deberán utilizar preferentemente las restricciones ya disponibles: PRIMARY KEY, FOREIGN KEY, CHECK, NOT NULL, UNIQUE, CASCADE, SET NULL, SET DEFAULT, etcétera.

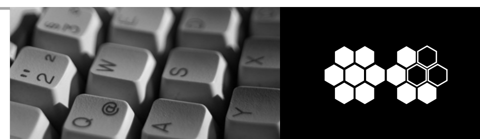
F. Activar, desactivar, compilar y eliminar disparadores

Un *trigger* puede estar activado o desactivado. Cuando se crea está activado, pero podemos variar esta situación mediante: `ALTER TRIGGER` *`nombretrigger`* `DISABLE`.

Para volver a activarlo utilizamos: `ALTER TRIGGER` *`nombretrigger`* `ENABLE`.

Para volver a compilar emplearemos: `ALTER TRIGGER` *`nombretrigger`* `COMPILE`.

Para eliminar un *trigger* escribiremos: `DROP TRIGGER` *`nombretrigger`*.



G. Vistas con información sobre los triggers

Las vistas `dba_triggers` y `user_triggers` contienen toda la información sobre los *trigger*. La estructura de estas vistas se muestran a continuación con algunos ejemplos.

```
SQL> describe dba_triggers
Name                           Null?       Type
----------------------------   --------    ---------------
OWNER                                      VARCHAR2(30)
TRIGGER_NAME                               VARCHAR2(30)
TRIGGER_TYPE                               VARCHAR2(16)
TRIGGERING_EVENT                           VARCHAR2(227)
TABLE_OWNER                                VARCHAR2(30)
BASE_OBJECT_TYPE                           VARCHAR2(16)
TABLE_NAME                                 VARCHAR2(30)
COLUMN_NAME                                VARCHAR2(4000)
REFERENCING_NAMES                          VARCHAR2(128)
WHEN_CLAUSE                                VARCHAR2(4000)
STATUS                                     VARCHAR2(8)
DESCRIPTION                                VARCHAR2(4000)
ACTION_TYPE                                VARCHAR2(11)
TRIGGER_BODY                               LONG

SQL> SELECT TRIGGER_NAME, TABLE_NAME, TRIGGERING_EVENT FROM
USER_TRIGGERS;


TRIGGER_NAME              TABLE_NAME     TRIGGERING_EVENT       TRIGGER_TYPE       STATUS
------------              ---------      ---------------        ------------       ------
AUDIT_SUBIDA_SALARIO      EMPLEADOS      UPDATE                 AFTER EACH ROW     ENABLED
T_GES_EMPLEAD             EMPLEAD        INSERT OR UPD...       INSTEAD OF         ENABLED


SQL> SELECT TRIGGER_BODY FROM USER_TRIGGERS;

TRIGGER_BODY
-------------------------------------------------------
BEGIN
  insert INTO auditaremple VALUES ('SUBIDA SALARIO EMPLEADO
' || :old.e…
  ...
DECLARE
  v_dept depart.dept_no%TYPE;
BEGIN
  IF DELETING THEN
  ...
```

11.3 Registros y colecciones

Los **registros** y las **colecciones** son estructuras de datos compuestas de otras más simples.

A. Registros en PL/SQL

El concepto de **registro** en PL/SQL es similar al de la mayoría de los lenguajes de programación (en C se llaman *estructuras*). Se trata de una estructura compuesta de otras más simples (*campos*) que pueden ser de distintos tipos.

A lo largo de este libro hemos venido utilizando el atributo ROWTYPE para crear una estructura de datos idéntica a la fila de una tabla, vista o cursor. Esta estructura de datos es un registro. Esta forma de crear registros es muy sencilla y rápida pero tiene algunas limitaciones:

- Tanto los nombres de los campos como sus tipos quedarán determinados por los nombres y los tipos de las columnas de la tabla, sin que exista posibilidad de variar alguno de ellos ni de excluir algunos campos o incluir otros.
- No se incluyen restricciones como NOT NULL ni valores por defecto.
- La posibilidad de creación de variables de registro utilizando este formato quedará condicionada a la existencia de la tabla de referencia y a sus posibles variaciones.

PL/SQL permite salvar estas limitaciones definiendo nosotros mismos el registro, y declarando después todas las variables de ese tipo que necesitemos. Para ello procederemos de este modo:

1. **Se define el tipo** genérico del registro:

```
TYPE nombre_tipo IS RECORD
(nombre_campo1 Tipo_campo1 [ [NOT NULL] {:= | DEFAULT}
  valorinicial1],
nombre_campo2 Tipo_campo2 [ [NOT NULL] {:= | DEFAULT}
  valorinicial2],
...);
```

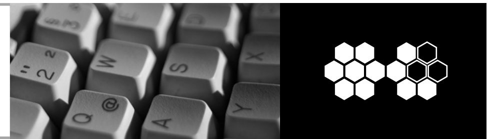
Al definir el tipo podemos usar el especificador NOT NULL. En ese caso los campos deben ser inicializados:

```
TYPE nombre_tipo IS RECORD
  (nombre_campo1 Tipo_campo1 NOT NULL  := valor1,
  nombre_campo2 Tipo_campo2 := valor2,
  ...);
```

Los tipos de los campos se pueden especificar también utilizando %TYPE y %ROWTYPE.

2. **Se declaran las variables** que se necesiten de ese tipo según el formato:

```
nombre_variable   nombre_tipo;
```



El siguiente ejemplo define el tipo **t_domicilio** y posteriormente declara la variable **v_domici** de ese tipo:

```
TYPE t_domicilio IS RECORD
  (calle VARCHAR2(30),
   numero SMALLINT,
   localidad VARCHAR2(25) );
v_domici t_domicilio;
```

Para referirnos a la variable de registro completa, indicaremos su nombre. Para referenciar solamente un campo, lo haremos utilizando la notación de punto según el formato:

```
nombre_variable_registro.nombre_campo
```

Podemos usar esta notación para asignar valores como si se tratase de cualquier otra variable. Pero debemos tener cuidado con expresiones del tipo `Nom_var_reg1 := Nom_var_reg2`, pues solamente funcionarán cuando ambas variables hayan sido definidas sobre el mismo tipo-base. En caso contrario, aun cuando coincidan los tipos, longitudes e, incluso, nombres de los campos, dará error.

Un campo de un registro puede ser, a su vez, un registro. En este caso se les llama **registros anidados.** En el siguiente ejemplo, al declarar el campo domicilio se indica que es de tipo **t_domicilio,** definido previamente y que incluye los campos *calle, numero* y *localidad*. Posteriormente, se podrá hacer referencia a estos campos utilizando la notación de punto, tal como aparece en el ejemplo:

```
DECLARE
    TYPE t_domicilio IS RECORD
       (calle VARCHAR2(30),
        numero SMALLINT,
        localidad VARCHAR2(25) );
    TYPE t_datospersona IS RECORD
        (nombre VARCHAR2(35),
        domicilio t_domicilio,
        fecha_nacimiento DATE);
    v_persona t_datospersona;
BEGIN
    v_persona.nombre := 'ALONSO FERNÁNDEZ, JOAQUÍN';
    v_persona.domicilio.calle := 'C/ ALBUFERA';
    v_persona.domicilio.numero := 14;
END;
```

Podemos definir tipos de registro, como objetos de la base de datos, aplicando el formato de definición de los registros junto con el de los objetos tal como se explica al final de esta unidad.

Los registros se pueden usar como parámetros y como valor de retorno de procedimientos y funciones. En estos casos, es conveniente definir el tipo base del registro externamente, de manera que esté accesible para todos los programas que vayan a usarlo.

Por ejemplo, en una declaración pública de un paquete o como un objeto de la base de datos. Ambos casos los estudiaremos más adelante en esta misma unidad.

B. Colecciones PL/SQL

Las **colecciones** son estructuras compuestas por listas de elementos. Se usan para guardar datos en formato de múltiples filas similar a las tablas de la base de datos.

Oracle dispone de tres tipos de colecciones:

- *Varrays.*
- Tablas anidadas.
- Tablas indexadas o *arrays* asociativos.

Todas ellas son listas de una dimensión aunque sus elementos pueden ser compuestos.

Varrays

Son equivalentes a los *arrays* (tablas de una dimensión) de los lenguajes de programación tradicionales.

- Pueden usarse en tablas de la base de datos.
- Se puede acceder a ellas desde SQL y desde PL/SQL.
- Tienen un índice secuencial que permite el acceso a sus elementos. A diferencia de otros lenguajes de programación (Java, C, etcétera), el índice comienza en uno.
- Tienen una longitud fija determinada en el momento de su creación.

Para poder usar el tipo VARRAY debemos:

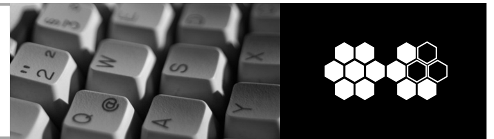
1. **Definir el tipo.** Podemos optar por definirlo a nivel de programa en la sección declarativa según el formato:

```
TYPE nombre_tipovarray IS VARRAY (numelementos) OF
tipoelementos [NOT NULL];
```

TYPE *nombre_tipovarray* IS VARRAY (*numelementos*) OF *tipoelementos* [NOT NULL];

- *nombre_tipovarray* es un especificador que indica el tipo base de la tabla y que posteriormente se utilizará para definir tablas.
- *tipo_elementos* especifica el tipo de los elementos que va a contener. Pueden ser tipos predefinidos de la base de datos o tipos definidos por el usuario.

También podemos definirlo como un **objeto de la base de datos** (especialmente si queremos que esté disponible para otros programas) usando el formato:



```
CREATE OR REPLACE
TYPE nombre_tipovarray AS VARRAY(numelementos) OF
tipoelementos [NOT NULL];
```

2. **Declarar variables** de ese tipo en la sección declarativa del programa:

```
nombre_variable   nombre_tipovarray;
```

3. **Inicializamos la variable** cargando valores. Esto podemos hacerlo:

   – En la misma declaración de la variable según el formato:

```
nombre_variable   nombre_tipovarray  :=
 nombre_tipovarray(lista de valores);
```

   – En la zona ejecutable haciendo asignaciones individuales:

```
nombre_variable := nombre_tipovarray (lista de valores);
```

4. **Para hacer referencia a los elementos** en el programa mediante el nombre de la variable y, entre paréntesis, el número del elemento.

```
nombre_variable(numelemento)
```

Al asignar una lista a una variable tipo VARRAY indicaremos el tipo sobre el que está definida la lista:

tipovarray(*lista*);

Los elementos deben ser inicializados antes de ser referenciados (aunque sea con valores nulos), pues de lo contrario nos encontraremos con el error:

```
ORA-06531: Reference to uninitialized collection.
```

Por ejemplo, podemos definir el tipo *t_meses* y declarar una variable de ese tipo incluyendo los meses:

```
DECLARE
    TYPE t_meses IS VARRAY (12) OF VARCHAR2(10);
    va_meses t_meses;
BEGIN
  va_meses := t_meses('ENERO', 'FEBRERO', 'MARZO', 'ABRIL',
'MAYO', 'JUNIO', 'JULIO', 'AGOSTO', 'SEPTIEMBRE',
          'OCTUBRE', 'NOVIEMBRE', 'DICIEMBRE');
    FOR i IN 1..12 LOOP
    DBMS_OUTPUT.PUT_LINE( TO_CHAR(i) || '-' || va_meses(i));
    END LOOP;
END;
/
```

El tipo sobre el que se define un VARRAY (u otra colección) puede ser a su vez un tipo definido por el usuario, frecuentemente un registro. En este caso, para referirnos a un campo elemento *i* de la variable lo haremos:

```
nobrevariablevarray(i).nombrecampo
```



Caso práctico

3 Escribiremos un bloque PL/SQL que realizará lo siguiente:

- Declarar un cursor basado en una consulta.
- Definir un tipo de registro compatible con el cursor.
- Definir un tipo de VARRAY cuyos elementos son del tipo registro previamente definido.
- Declarar inicializar y usar una variable de tipo VARRAY cargando el contenido del cursor en los elementos y posteriormente mostrando el contenido de estos.

```
DECLARE
   /* Declaramos un cursor basado en una consulta */
    CURSOR c_depar IS
        SELECT dnombre, count(emp_no) numemple
        FROM depart, emple
        WHERE depart.dept_no = emple.dept_no
        GROUP BY depart.dept_no, dnombre;

    /* Definimos un tipo compatible con el cursor */
     TYPE tr_depto IS RECORD (
        nombredep depart.dnombre%TYPE,
        numemple INTEGER
        );

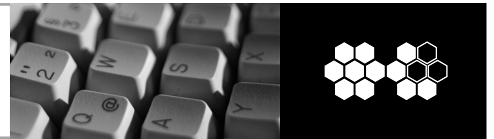
  /* Definimos un tipo VARRAY basado en el tipo anterior */
  TYPE tv_depto IS VARRAY (6) OF tr_depto;

  /* Declaramos e inicializamos una variable del tipo VARRAY definido arriba */
  va_departamentos tv_depto := tv_depto(NULL,NULL,NULL,NULL,NULL,NULL);

  /* Declaramos una variable para usarla como índice */
  n INTEGER := 0;
BEGIN

   /* Cargar valores en la variable */
  FOR vc IN c_depar LOOP
        n := c_depar%ROWCOUNT;
        va_departamentos(n) := vc;
    END LOOP;

    /* Mostrar los datos de la variable */
    FOR i IN 1..n LOOP
        DBMS_OUTPUT.PUT_LINE(' * Dnombre:'  || va_departamentos(i).nombredep ||
                             ' * NºEmpleados: ' || va_departamentos(i).numemple );
    END LOOP;
 END;
   /
```



Tablas anidadas PL/SQL

Son estructuras similares a los VARRAYS estudiados en el epígrafe anterior y comparten con ellos muchas características estructurales y funcionales (lista de elementos, índice para acceso, acceso desde SQL y PL/SQL, posibilidad de uso en tablas de la base de datos, etcétera).

Pero también existen importantes diferencias, por ejemplo, las tablas anidadas no tienen una longitud fija. Para crearlas:

1. **Definimos el tipo,** a nivel de programa, en la sección declarativa según el formato:

```
TYPE nombre_tipoTablaAnidada IS TABLE OF tipoelementos
[NOT NULL];
```

También podemos definirlo como un objeto de la base de datos usando el formato:

```
CREATE OR REPLACE
TYPE nombre_tipoTablaAnidada AS TABLE OF tipoelementos
[NOT NULL];
```

2. **Declaramos e inicializamos las variables,** igual que en el caso de los VARRAY, pero en este caso las tablas anidadas permiten inicializar la variable con una lista incompleta o vacía:

```
nombre_variable   :=  nombre_tipovarray();
```

Y añadir filas nuevas a la variable ya inicializada usando el método EXTEND según el formato:

```
nombre_variable_de_tabla_anidada.EXTEND;
```

El siguiente ejemplo ilustra las similitudes y diferencias de las tablas anidadas con los VARRAY y el uso del método EXTEND.

```
DECLARE
    TYPE t_meses IS TABLE OF VARCHAR2(10);
    va_meses t_meses;
BEGIN
   va_meses := t_meses('ENERO', 'FEBRERO', 'MARZO', 'ABRIL',
        'MAYO', 'JUNIO', 'JULIO', 'AGOSTO', 'SEPTIEMBRE',
        'OCTUBRE');
    va_meses.EXTEND;
    va_meses(11):= 'NOVIEMBRE';
    va_meses.EXTEND;
    va_meses(12):= 'DICIEMBRE';
    FOR i IN 1..12 LOOP
    DBMS_OUTPUT.PUT_LINE( TO_CHAR(i) || '-' || va_meses(i));
    END LOOP;
END;
/
```

El **tipo de dato TABLE** en PL/SQL sirve para definir estructuras de datos de tipo array. Es completamente distinto del tipo TABLE que utilizan los SGBDR para almacenar la información.



Actividades propuestas

3 **Reescribe el bloque PL/SQL del caso práctico del epígrafe anterior usando una tabla anidada en lugar de un VARRAY. Debemos tener presente que no es necesario inicializar a NULL varios elementos, sino inicializar con una lista vacía y, después, podemos crear nuevos elementos en el bucle de carga usando el método EXTEND.**

Las tablas anidadas y los VARRAYS permiten su uso en tablas de la base de datos y su manejo mediante instrucciones SQL. Ambos extremos exceden nuestros objetivos, pero remitimos a la documentación del producto para ampliaciones sobre esta materia.

El siguiente tipo de colección que estudiaremos sólo permite su uso desde PL/SQL.

Tablas indexadas (o *arrays* asociativos)

Las tablas indexadas han evolucionado de la estructura conocida como TABLAS-PL/SQL de las versiones de Oracle 7 y 8.

Son estructuras tipo *array,* similares a las anteriores pero con diferencias importantes:

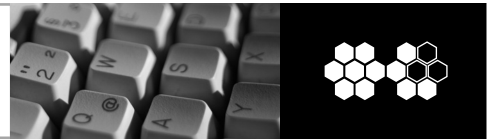
- No pueden usarse en tablas de la base de datos.
- No pueden manipularse con comandos SQL, sólo con PL/SQL.
- No son objetos.
- No tienen una longitud predeterminada.
- No requieren inicialización.
- Todos los elementos se crean dinámicamente.
- El índice no es secuencial, suele ser de tipo BINARY_INTEGER o PLS_INTEGER y puede tomar cualquier valor de los permitidos por estos tipos (positivo, negativo o cero).

Son tablas exclusivas de PL/SQL cuyos elementos se crean dinámicamente y cuyo índice no es secuencial. Tienen funcionalidades similares a las listas enlazadas.

Las operaciones a realizar para usar estas tablas son:

1. **Definir el tipo base de la tabla.** Igual que en los anteriores pero en este caso no indicaremos número de elementos y sí el tipo de índice.

```
TYPE nombre_tipoTabla IS TABLE OF tipo_elementos
[NOT NULL]
INDEX BY [PLS_INTEGER | BINARY_INTEGER | VARCHAR2(longitud)];
```



También pueden crearse como un tipo de la base de datos mediante el comando CREATE OR REPLACE TYPE …

2. **Declarar variables** de ese tipo utilizando el formato:

nombrevariableTabla nombre_tipoTabla;

En el caso de las tablas indexadas, las variables no requieren inicialización y tampoco hay que reservar memoria para los nuevos elementos, simplemente introduciremos un valor indicando el índice del elemento.

3. Para **hacer referencia** a los elementos en el programa mediante el nombre de la variable y, entre paréntesis, el número del elemento: *nombre_variable(indice)*.

```
DECLARE
  TYPE T_tabla_emple IS TABLE OF emple%ROWTYPE;
    INDEX BY BINARY_INTEGER;
  tab_emple T_tabla_emple;
  ...
BEGIN
  ...
  SELECT * INTO tab_emple(7900)
    WHERE emp_no = 7900;
  …
  DBMS_OUTPUT.PUT_LINE(tab_emple(7900).apellido);
  …
  tab_emple(7900).salario := 3500;
  …
```

Debemos recordar que los elementos de una tabla indexada no tienen que ser necesariamente contiguos.

Cuando los elementos de la tabla son de tipo registro utilizaremos la notación de punto según el siguiente formato:

Nombretabla (indiceelemen). nombrecampo.

Los *arrays* indexados son muy útiles para cargar datos de una tabla de la base de datos usando como índice la clave primaria.

```
DECLARE
    TYPE T_tabla_emple IS TABLE OF emple%ROWTYPE
      INDEX BY BINARY_INTEGER;
    tab_emple T_tabla_emple;
    CURSOR c_emple IS SELECT * FROM emple;
    ...
BEGIN
    .../*El sigte. bucle carga en la tabla las filas de
emple*/
    FOR v_reg_emple IN c_emple LOOP
        tab_emple(v_reg_emple.emp_no):= v_reg_emple;
    END LOOP;
    ...
```

La flexibilidad del índice no secuencial plantea también un problema a la hora de recorrer la tabla. Oracle dispone de una API para el manejo de colecciones que facilita esta tarea.

Atributos de colecciones PL/SQL

Facilitan la gestión de las variables de colecciones permitiendo recorrer la tabla, contar y borrar los elementos, etcétera. En general, para utilizar los atributos se empleará el siguiente formato:

variabledetabla.atributo[(listadeparámetros)]

Los parámetros hacen referencia, normalmente, a valores de índice:

- FIRST. Devuelve el valor de la clave o índice del primer elemento de la tabla:

  ***variabletabla*.FIRST**

- LAST. Devuelve el valor de la clave o índice del último elemento de la tabla:

  ***variabletabla*.LAST**

  Por ejemplo, para recorrer una tabla cuyo índice sabemos que tiene valores consecutivos podemos escribir:

```
FOR i IN  variabletabla.FIRST .. variabletabla.LAST LOOP
```

- PRIOR. Devuelve el valor de la clave o índice del elemento anterior al elemento ***n:***

  ***variabletabla*.PRIOR(n)**

- NEXT. Devuelve el valor de la clave o índice del elemento posterior al elemento ***n:***

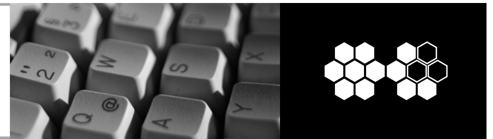
  ***variabletabla*.NEXT(n)**

  PRIOR y NEXT se pueden utilizar para recorrer una tabla (en cualquiera de los dos sentidos) incluso con valores de índice no consecutivos. No obstante, deberemos tener cuidado con los valores que devolverán en ambos extremos, ya que PRIOR del primer elemento devuelve NULL y lo mismo ocurre con NEXT del último elemento.

```
...
i:= variabletabla.FIRST
WHILE i IS NOT NULL LOOP
  ...
  i:= variabletabla.NEXT(i);
END LOOP;
...
```

- COUNT. Devuelve el número de filas que tiene una tabla:

  ***variabletabla*.COUNT**



- EXISTS. Devuelve TRUE si existe el elemento *n;* en caso contrario devolverá FALSE. Se utiliza para evitar el error que se produce cuando intentamos acceder a un elemento que no existe en la tabla:

  ***variabletabla*.EXISTS(n)**

- DELETE. Se utiliza para borrar elementos de una tabla:

  - ***variabletabla*.DELETE.** Borra todos los elementos de la tabla.
  - ***variabletabla*.DELETE(*n*).** Borra el elemento indicado por *n*. Si el valor de *n* es NULL no hará nada.
  - ***variabletabla*.DELETE(*n1, n2*).** Borra las filas comprendidas entre *n1* y *n2,* siendo *n1*>=*n2* (en caso contrario no hará nada).

Todos estos atributos están disponibles para todas las colecciones (VARRAYS, TABLAS ANIDADAS y TABLAS INDEXADAS); pero además existen otros que sólo están disponibles para alguna de las dos primeras:

- EXTEND. Reserva espacio para un nuevo elemento (VARRAYS y TABLAS ANIDADAS). Opcionalmente puede incluirse un parámetro que indique el número de elementos nuevos (por defecto se asume uno):

  ***variabletabla*.EXTEND;** o bien ***variabletabla*.EXTEND(n);**

  El atributo EXTEND se puede usar también en los VARRAYS, siempre que no se exceda el límite indicado en la declaración (por ejemplo, cuando se inicializó con menos elementos de los previstos en la declaración) pues de lo contrario dará error.

- TRIM. Elimina el último elemento (el índice más alto en VARRAYS y TABLAS ANIDADAS). Opcionalmente puede incluirse un parámetro que indique el número de elementos a eliminar (comenzando en el último, penúltimo, etcétera):

  ***variabletabla*.TRIM;** o bien ***variabletabla*.TRIM(n);**

- LIMIT. Devuelve el valor más alto permitido en un VARRAY:

  ***variabletabla*.LIMIT**



Caso práctico

4 **A continuación reescribiremos el código del Caso práctico 3 usando una tabla indexada y los atributos disponibles para recorrer la tabla.**

```
DECLARE
     CURSOR c_depar IS           /* Declaramos un cursor basado en una consulta */
        SELECT depart.dept_no, dnombre, count(emp_no) numemple
        FROM depart, emple
        WHERE depart.dept_no = emple.dept_no
        GROUP BY depart.dept_no, dnombre;

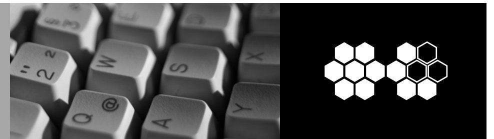
     TYPE tr_depto IS RECORD (     /* Definimos un tipo compatible con el cursor */
        nombredep depart.dnombre%TYPE,
        numemple INTEGER);

               /* Definimos un tipo TABLA INDEXADA basado en el tipo anterior */
     TYPE ti_depto IS TABLE OF tr_depto INDEX BY PLS_INTEGER;

     va_departamentos ti_depto; /* Declaramos la variable del tipo TABLA INDEXADA */

     n PLS_INTEGER := 0;       /* Declaramos una variable para usarla como índice */
BEGIN
     FOR vc IN c_depar LOOP             /* Cargar valores. El indice es el NºDpto */
        va_departamentos(vc.dept_no).nombredep := vc.dnombre;
        va_departamentos(vc.dept_no).numemple := vc.numemple;
     END LOOP;

     n := va_departamentos.FIRST;
     WHILE va_departamentos.EXISTS(n) LOOP    /* Mostrar los datos de la variable */
        DBMS_OUTPUT.PUT_LINE(' * Dep Nº :'  || n ||
                             ' * Dnombre:'  || va_departamentos(n).nombredep ||
                             ' * NºEmpleados: ' || va_departamentos(n).numemple );
        n := va_departamentos.NEXT(n);
     END LOOP;
END;
/
```



11.4 Paquetes

Se utilizan para agrupar y guardar programas y otros objetos en la base de datos. En un entorno de producción, todos los programas y objetos (procedimientos, funciones cursores, etcétera) de una determinada aplicación o que gestionen un área de negocio, se agruparán en un paquete o en varios. También se pueden crear paquetes con utilidades que den soporte a distintas aplicaciones.

Oracle dispone de paquetes predefinidos (DBMS_OUTPUT, DBMS_STANDARD, ...) donde se encuentran muchas de las funciones y utilidades que hemos venido utilizando hasta ahora.

A. Elementos de un paquete

En los paquetes hay dos elementos claramente diferenciados:

- **Especificación:** contiene declaraciones públicas de subprogramas, tipos, constantes, variables, cursores, excepciones, etcétera. Los objetos declarados en la especificación son accesibles también desde fuera del paquete. Actúa como una interfaz con el exterior.

- **Cuerpo:** contiene los detalles de implementación de todos los objetos del paquete (el código de los programas, etcétera). También puede incluir declaraciones de objetos accesibles solamente desde los objetos del paquete.

B. Creación de un paquete

Tanto la especificación como el cuerpo se pueden crear desde SQL*PLUS mediante los comandos CREATE [OR REPLACE] PACKAGE y CREATE [OR REPLACE] PACKAGE BODY, respectivamente. El formato genérico es:

- **Creación de cabecera o especificación:**

```
CREATE [OR REPLACE] PACKAGE nombredepaquete AS
<declaraciones de tipos, variables, cursores, excepciones
y otros objetos públicos>
<especificación de subprogramas>
END [nombredepaquete];
```

- **Creación del cuerpo del paquete:**

```
CREATE [OR REPLACE] PACKAGE BODY mombredepaquete AS
<declaraciones de tipos, variables, cursores, excepciones
y otros objetos privados>
      <cuerpo de los subprogramas>
[BEGIN
      <instrucciones iniciales>
END [nombredepaquete];
```

El siguiente ejemplo servirá para ilustrar la utilización de paquetes en PL/SQL. En primer lugar se crea la cabecera o especificación del paquete según el formato indicado.

```
/* cabecera o especificación del paquete */
CREATE OR REPLACE PACKAGE buscar_emple
AS
    TYPE t_reg_emple IS RECORD
     (num_empleado emple.emp_no%TYPE,
      apellido emple.apellido%TYPE,
      oficio emple.oficio%TYPE,
      salario emple.salario%TYPE,
      departamento emple.dept_no%TYPE);

    PROCEDURE ver_por_numero
      (v_emp_no emple.emp_no%TYPE);

    PROCEDURE ver_por_apellido
      (v_apellido emple.apellido%TYPE);

    FUNCTION datos
     (v_emp_no emple.emp_no%TYPE)
     RETURN t_reg_emple;

END buscar_emple;
/
```

El sistema responderá con el mensaje: `PAQUETE CREADO`.

O bien: `Aviso: Paquete creado con errores de compilación.`

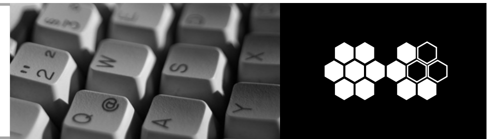
En este caso habrá que depurar los posibles errores antes de proceder a crear el cuerpo del paquete.

Declaración de objetos en la cabecera o especificación del paquete

Podemos observar en el ejemplo que se han declarado:

- Un tipo: `TYPE t_reg_emple`
- Dos procedimientos: `PROCEDURE ver_por_numero`
  `PROCEDURE ver_por_apellido`
- Una función: `FUNCTION datos`

Las declaraciones de procedimientos y funciones en la cabecera deben ser ***declaraciones formales,*** es decir, contendrán únicamente la cabecera de los subprogramas, el nombre, la definición de los parámetros y el tipo de retorno en las funciones.



Todos los objetos declarados en la cabecera del paquete son accesibles tanto desde el propio paquete como desde el exterior. En este último caso se deberá utilizar el nombre del objeto precedido por el nombre del paquete utilizando la notación de punto:

Nombre_de_paquete.nombre_de_objeto

Por ejemplo, para ejecutar el procedimiento *ver_por_numero* desde SQL*Plus escribiremos:

```
SQL> execute buscar_emple.ver_por_numero(7902);
```

Se pueden utilizar, aplicando la notación indicada, los otros objetos declarados en la cabecera. Por ejemplo, se puede utilizar desde otro programa el tipo *t_reg_emple* para definir datos:

```
DECLARE
...
   Mi_empleado t_reg_emple;
...
```

Creación del cuerpo del paquete y declaración de objetos locales

Una vez creada la cabecera crearemos el cuerpo del paquete, el cual:

- Incluirá el código correspondiente a las declaraciones formales realizadas en la cabecera.

- Podrá incluir otros objetos locales al paquete: tipos, variables, excepciones, procedimientos, funciones, etcétera. Estos objetos serán accesibles desde cualquier parte del paquete, pero no desde fuera.

```
/* Cuerpo del paquete */
CREATE OR REPLACE PACKAGE BODY buscar_emple
AS

  vg_emple t_reg_emple;  /* variable local al paquete */
  PROCEDURE ver_emple;   /* declaración del procedimiento
                            local al paquete */
  PROCEDURE ver_por_numero
      (v_emp_no emple.emp_no%TYPE)
  IS
  BEGIN
      SELECT emp_no, apellido, oficio, salario, dept_no
        INTO vg_emple
        FROM emple
        WHERE emp_no = v_emp_no;
        ver_emple; /* llama al procedimiento ver_emple */
  END ver_por_numero;
```

> Para poder crear el cuerpo del paquete deberemos haber creado previamente la cabecera.

```
PROCEDURE ver_por_apellido
     (v_apellido emple.apellido%TYPE)
  IS
  BEGIN
      SELECT emp_no, apellido, oficio, salario, dept_no
        INTO vg_emple
        FROM emple
        WHERE apellido = v_apellido;
      ver_emple;
  END ver_por_apellido;

  FUNCTION datos
    (v_emp_no emple.emp_no%TYPE)
  RETURN t_reg_emple
  IS
  BEGIN
      SELECT emp_no, apellido, oficio, salario, dept_no
       INTO vg_emple
       FROM emple
       WHERE emp_no = v_emp_no;
      RETURN vg_emple;
  END datos;

  PROCEDURE ver_emple     /* Implementación del procedi-
miento*/
  IS
  BEGIN
      DBMS_OUTPUT.PUT_LINE(vg_emple.num_empleado||
      '*'||vg_emple.apellido||'*'||vg_emple.oficio ||
      '*'||vg_emple.salario||'*'||vg_emple.departamento);

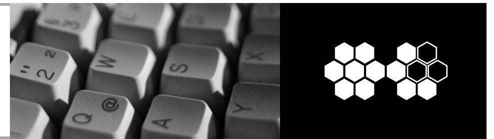
  END ver_emple;

END buscar_emple;
/
```

El sistema responderá con el mensaje: **CUERPO DEL PAQUETE CREADO.**

Hemos de tener en cuenta que no se podrá compilar el cuerpo del paquete hasta que se haya creado correctamente la cabecera, ya que, de lo contrario, se producirá el siguiente error:

```
LINE/COL ERROR
----- -------------------------------------------------
0/0   PL/SQL: Compilation unit analysis terminated
1/14 PLS-00905: el objeto FERNANDO.BUSCAR_EMPLE no es
válido
1/14  PLS-00304: no se puede compilar el cuerpo de 'BUS-
CAR_EMPLE'  sin su especificación
```



En nuestro ejemplo, además del código correspondiente a los subprogramas declarados en la cabecera, el cuerpo del procedimiento incluye:

- Una variable: `vg_emple` de tipo *`t_reg_emple;`*
- Un procedimiento: `PROCEDURE ver_emple;`

C. Utilización de los objetos definidos en el paquete

- **Desde el mismo paquete.** Todos los objetos declarados en el paquete pueden ser utilizados por los demás objetos del mismo, con independencia de que hayan sido o no declarados en la especificación. Además, no necesitan utilizar la notación de punto para referirse a los objetos del mismo paquete.

  Los procedimientos `ver_por_numero` y `ver_por_apellido` llaman al procedimiento *`ver_emple`*, que no está en la especificación. Asimismo, utilizan la variable *`vg_emple`*, que tampoco se encuentra en la especificación.

- **Desde fuera del paquete.** Los subprogramas y otros objetos contenidos en el paquete son accesibles desde fuera solamente si han sido declarados en la especificación. En nuestro ejemplo no se podría hacer referencia desde fuera del paquete al procedimiento *`ver_emple`*, ya que no ha sido declarado en la cabecera.

Para ejecutar un procedimiento de un paquete desde SQL*Plus se utilizará el formato:

```
SQL> EXECUTE nombrepaquete.nomsubprog(listaparam);
```

Solamente se podrán ejecutar desde fuera los subprogramas que hayan sido declarados en la especificación.

A continuación, veremos dos ejemplos de ejecución:

```
SQL> SET SERVEROUTPUT ON
SQL>  EXECUTE buscar_emple.ver_por_numero(7902);
7902*FERNANDEZ*ANALISTA*3900*20

SQL>  EXECUTE buscar_emple.ver_por_apellido('JIMENO');
7900*JIMENO*EMPLEADO*1235*30
```

Se pueden utilizar los subprogramas declarados en la especificación desde otros subprogramas que se encuentran fuera del paquete. El siguiente procedimiento visualiza el apellido, el salario y el oficio de un empleado cuyo número se pasa en la llamada, haciendo uso del tipo `buscar_emple.t_reg_emple` y de la función `buscar_emple.datos`:

```
CREATE OR REPLACE PROCEDURE ver_datos_empleado
    (v_num_emple emple.emp_no%TYPE)
AS
    vr_emple buscar_emple.t_reg_emple;
```

```
BEGIN
    vr_emple := buscar_emple.datos(v_num_emple);
    DBMS_OUTPUT.PUT_LINE(vr_emple.apellido ||'*'||
                         vr_emple.salario||'*'||
                         vr_emple.oficio);
END ver_datos_empleado;
```

D. Declaración de cursores en paquetes

Para declarar cursores en paquetes, si queremos que estén accesibles desde fuera, deberemos separar la declaración del cursor del cuerpo (en éste es donde va la cláusula SELECT).

La declaración del cursor se incluirá en la cabecera del paquete (para que pueda estar accesible desde fuera del paquete) indicando el nombre del cursor, los parámetros (si procede) y el tipo devuelto. Este último se indicará mediante RETURN ***tipodedato;*** para cursores que devuelven filas enteras normalmente se usará %ROWTYPE.

```
CREATE PACKAGE empleados_acc AS
   ...
   CURSOR c1 RETURN emple%ROWTYPE;
   ...
END empleados_acc;


CREATE PACKAGE BODY empleados_acc AS
   ...
   CURSOR c1 RETURN emple%ROWTYPE
    SELECT * FROM emple WHERE salario > 1000;
   ...
END empleados_acc;
```

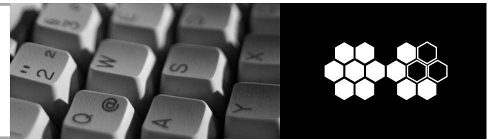
E. Ámbito y otras características de las declaraciones

Se pueden utilizar paquetes exclusivamente para declarar tipos, constantes, variables, excepciones, cursores, etcétera. En estos casos no es necesario el cuerpo del paquete.

Los objetos declarados en la especificación son accesibles desde el propio paquete y también desde fuera (en este caso, usando la notación de punto). Además, todas las variables declaradas en la especificación del paquete mantienen su valor durante la sesión, por tanto, el valor no se pierde entre las llamadas de los subprogramas.

Respecto a las variables constantes y cursores declarados en un paquete:

- El propietario es la sesión.
- En la sesión no se crean los objetos hasta que se referencia el paquete.
- Cuando se crean los objetos su valor será nulo (salvo que se inicialice).
- Durante la sesión los valores pueden cambiarse.
- Al salir de la sesión los valores se pierden.



F. Características de almacenamiento y compilación

Tanto el código fuente como el código compilado de los paquetes se almacena en la base de datos. Al igual que ocurría con los subprogramas almacenados, el *paquete* (la especificación) puede tener dos estados:

- Disponible (*valid*).
- No disponible (*invalid*).

Cuando se borra o modifica alguno de los objetos referenciados el paquete pasará a *invalid*. Si la especificación del paquete se encuentra «no disponible», Oracle invalida (pasa a *invalid*) cualquier objeto que haga referencia al paquete.

Cuando recompilamos la especificación, todos los objetos que hacen referencia al paquete pasan a «no disponible» hasta que sean compilados de nuevo. Cualquier referencia o llamada a uno de estos objetos antes de ser recompilados producirá que Oracle automáticamente los recompile. Esto ocurre también con el cuerpo del paquete, pero en este caso se puede recompilar el paquete sin invalidar la especificación. Esto evita las recompilaciones en cascada innecesarias.

Si se cambia la definición o implementación (*cuerpo*) de una función o procedimiento incluido en un paquete, no hay que recompilar todos los programas que llaman al subprograma (como ocurre con los subprogramas almacenados), a no ser que también se cambie la especificación de dicha función o procedimiento.

G. Paquetes suministrados por Oracle

Oracle incluye con su gestor de bases de datos diversos paquetes como STANDARD, DBMS_OUTPUT, DBMS_STANDARD, DBMS_SQL, y muchos otros que incorporan diversas funcionalidades. A continuación, veremos una breve reseña del contenido de estos paquetes:

- En STANDARD se declaran tipos, excepciones, funciones, etcétera, disponibles desde el entorno PL/SQL. Por ejemplo, en el paquete STANDARD están definidas las funciones:

```
FUNCTION ABS (n number) RETURN NUMBER;
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VAR-
   CHAR2;
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN
   VARCHAR2;
```

  Si, por alguna circunstancia, volvemos a declarar alguna de esas funciones desde un programa PL/SQL, siempre podremos hacer referencia a la original mediante la notación de punto. Por ejemplo, `... STANDARD.ABS(...) ...`

Todos los paquetes suministrados por Oracle están documentados en los manuales del producto.

- DBMS_STANDARD incluye utilidades, como el procedimiento RAISE_APLICATION_ERROR, que facilitan la interacción de nuestras aplicaciones con Oracle.

- En DBMS_OUTPUT se encuentra el procedimiento PUT_LINE que hemos venido utilizando para visualizar datos, y otros como ENABLE y DISABLE, que permiten configurar y purgar el buffer utilizado por PUT_LINE.

- DBMS_SQL incorpora procedimientos y funciones que permiten utilizar SQL dinámico en nuestros programas, tal como veremos en el apartado siguiente.

11.5 SQL Dinámico

Cuando se compila un procedimiento, PL/SQL comprueba en el diccionario de datos y completa todas las referencias a objetos de la base de datos (tablas, columnas, usuarios, etcétera). De esta forma, si algún objeto no existe en el momento de la compilación, el proceso fallará, indicando el error correspondiente.

Esta manera de trabajar se denomina **SQL estático** y tiene importantes ventajas, especialmente en cuanto a velocidad en la ejecución y a la seguridad de que las referencias a los objetos han sido comprobadas previamente; pero también tiene algunos inconvenientes:

- Todos los objetos tienen que existir en el momento de la compilación.

- No se pueden crear objetos nuevos con el programa.

- No se puede cambiar la definición de los objetos.

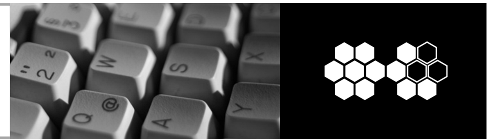
- Cualquier referencia a un objeto debe ser conocida y resuelta en tiempo de compilación.

Existe la posibilidad de superar estas limitaciones y ejecutar instrucciones de definición de datos, así como resolver referencias a objetos en el momento de la ejecución. A esta forma de trabajar se le denomina **SQL dinámico.**

Hasta la versión 9i la única posibilidad de trabajar con SQL dinámico era usar el paquete **DBMS_SQL**. A partir de esta versión Oracle incorpora la posibilidad de trabajar con ***SQL dinámico nativo (NDS)***. Nosotros estudiaremos las dos posibilidades pues, aunque la tendencia actual es trabajar con NDS, hay mucho código desarrollado con DBMS_SQL y es la única posibilidad soportada en instalaciones anteriores.

A. SQL dinámico con DBMS_SQL

El siguiente programa es capaz de ejecutar órdenes de definición y manipulación sobre objetos que sólo se conocerán al ejecutar el programa, ya que el comando se pasará en la llamada al procedimiento:



```
CREATE OR REPLACE PROCEDURE ejsqldin
  (instruccion VARCHAR2)
AS
  id_cursor INTEGER;
  v_dummy INTEGER;
BEGIN
  id_cursor :=  DBMS_SQL.OPEN_CURSOR;       /*Abrir*/
  DBMS_SQL.PARSE(id_cursor, instruccion, DBMS_SQL.V7);
/*Analizar*/
  v_dummy := DBMS_SQL.EXECUTE(id_cursor);  /*Ejecutar*/
  DBMS_SQL.CLOSE_CURSOR(id_cursor);         /*Cerrar*/
EXCEPTION
  WHEN OTHERS THEN
    DBMS_SQL.CLOSE_CURSOR(id_cursor);       /*Cerrar*/
    RAISE;
END ejsqldin;
/
```

En este ejemplo podemos observar resaltadas las líneas correspondientes a las cuatro operaciones más frecuentes que se suelen realizar al programar con SQL dinámico, así como los subprogramas incluidos en DBMS_SQL que se encargan de realizar dichas operaciones:

- DBMS_SQL.OPEN_CURSOR abre el cursor y devuelve un número de identificación para poder utilizarlo. Recordemos que todos los comandos SQL se ejecutan dentro de un cursor.

- DBMS_SQL.PARSE analiza la instrucción que se va a ejecutar. Este paso es necesario, ya que cuando se compiló el programa, la instrucción no pudo ser analizada, pues aún no se había construido.

- DBMS_SQL.EXECUTE se encarga de la ejecución.

- DBMS_SQL.CLOSE_CURSOR cierra el cursor utilizado.

Debemos tener en cuenta que, cuando se definen objetos dinámicamente, el propietario del procedimiento deberá tener concedidos los privilegios necesarios (CREATE USER, CREATE TABLE, etcétera) de forma directa, no mediante un rol.

```
SQL>  EXECUTE EJSQLDIN('CREATE USER DUMM1 IDENTIFIED BY
DUMM1');
begin EJSQLDIN('CREATE USER DUMM1 IDENTIFIED BY DUMM1');
end;
*
ERROR en línea 1:
ORA-01031: privilegios insuficientes
ORA-06512: en "SYSTEM.EJSQLDIN", línea 14
ORA-06512: en línea 1
```

Observemos que se trata del usuario SYSTEM, pero tiene el privilegio CREATE USER mediante un rol. Para solucionar el problema se concede el privilegio de forma directa:

```
SQL> GRANT CREATE USER TO SYSTEM;
Concesión terminada con éxito.
SQL>  EXECUTE EJSQLDIN('CREATE USER DUMM1 IDENTIFIED BY
DUMM1');
Procedimiento PL/SQL terminado con éxito.
```

A continuación, podemos observar otros ejemplos de ejecución:

```
SQL> EXECUTE EJSQLDIN('CREATE TABLE PR1 (C1 CHAR)');
Procedimiento PL/SQL terminado con éxito.

SQL> EXECUTE EJSQLDIN('ALTER TABLE PR1 ADD COMENTARIO
     VARCHAR2(20)')
Procedimiento PL/SQL terminado con éxito.

SQL> DESCRIBE PR1
 Name                    Null?  Type
 --------------------    -----  ----
 C1                      CHAR(1)
 COMENTARIO              VARCHAR2(20)
```

Pasos para utilizar SQL Dinámico con DBMS_SQL

Aunque hay diferencias dependiendo del tipo de instrucción que se va a procesar, los pasos que hay que seguir son los siguientes:

1. Abrir el cursor (OPEN_CURSOR) y guardar su número de identificación para posteriores referencias.

2. Analizar (PARSE) el comando que se va a procesar.

3. Si se van a pasar valores al comando, acoplar las variables de entrada (BIND).

4. Si el comando es una consulta:

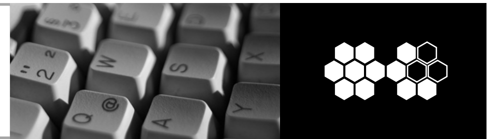
   – Definir columnas (DEFINE_COLUMN).

   – Ejecutar (EXECUTE).

   – Recuperar los valores con FETCH_ROWS y COLUMN_VALUE.

5. Si se trata de un comando de manipulación, ejecutar (EXECUTE).

6. Cerrar el cursor (CLOSE_CURSOR).



Comandos de definición de datos con DBMS_SQL

Utilizando los procedimientos y funciones incluidos en el paquete DBMS_SQL podemos crear objetos en tiempo de ejecución. Por ejemplo, el siguiente procedimiento creará una tabla de una sola columna cuyos datos se pasarán en la llamada al procedimiento:

```
CREATE OR REPLACE PROCEDURE creartabla
    (nombretabla VARCHAR2,
    nombrecol VARCHAR2,
    longitudcol POSITIVE)
AS
    id_cursor INTEGER;
    v_comando VARCHAR2(2000);
BEGIN
 id_cursor := DBMS_SQL.OPEN_CURSOR;  /*Abre el cursor*/
 v_comando := 'CREATE TABLE '|| nombretabla || ' ('
   || nombrecol || ' VARCHAR2(' || longitudcol || '))';
/*Analiza y ejecuta*/
DBMS_SQL.PARSE(id_cursor, v_comando, DBMS_SQL.NATIVE);
DBMS_SQL.CLOSE_CURSOR(id_cursor);  /*Cierra el cursor*/
EXCEPTION
  WHEN OTHERS THEN
   DBMS_SQL.CLOSE_CURSOR(id_cursor); /*Cierra el cursor*/
    RAISE;
END creartabla;
/
```

- OPEN_CURSOR. Esta función abre un cursor nuevo y retorna un identificador del mismo, que se guarda en una variable y servirá para hacer referencia al cursor desde el programa.

- PARSE. Analiza el comando comprobando posibles errores. Si se trata de un comando DDL (como en este caso), PARSE, además, lo ejecuta. Recibe tres parámetros:

  – El identificativo del cursor.

  – La cadena que contiene el comando (sin el ;).

  – El indicador del lenguaje, que determina la manera en que Oracle manejará el comando SQL. Están disponibles las siguientes opciones:

    - V7: estilo Oracle versión 7 (es el mismo para la versión 8 y la 9).

    - NATIVE: estilo por defecto de la base de datos en la que esté conectada la sesión.

- CLOSE_CURSOR. Cierra el cursor especificado y libera la memoria utilizada por el cursor.

Los objetos se crearán por defecto en el esquema del usuario propietario del procedimiento. Si se pretende crearlos en otro esquema, se deberá indicar explícitamente.

> Al utilizar SQL Dinámico aumentan las posibilidades de que aparezcan errores durante la ejecución del programa. Por eso, es muy importante incluir un manejador WHEN OTHERS que cierre el cursor.

Comandos de manipulación de datos con DBMS_SQL

También se pueden crear dinámicamente *instrucciones de manipulación de datos.*

El siguiente ejemplo muestra la utilización de PL/SQL Dinámico para introducir datos en una tabla que se indicará en la llamada (las llamadas a PUT_LINE se utilizan para ilustrar el funcionamiento del programa):

```
CREATE OR REPLACE PROCEDURE introducir_fila
    (nombretabla VARCHAR2,
    valor VARCHAR2)
AS
    id_cursor INTEGER;
    v_comando VARCHAR2(2000);
    v_filas INTEGER;
BEGIN
    id_cursor :=       DBMS_SQL.OPEN_CURSOR;
    v_comando := '     INSERT INTO ' || nombretabla || '
VALUES (:val_1)';
   DBMS_OUTPUT.PUT_LINE(v_comando);
   DBMS_SQL.PARSE(id_cursor, v_comando, DBMS_SQL.NATIVE);
   DBMS_SQL.BIND_VARIABLE(id_cursor, ':val_1', valor);
/*Acopla la variable que pasará el valor de entrada*/
   v_filas := DBMS_SQL.EXECUTE(id_cursor); /*Ejecuta el
comando*/
    DBMS_SQL.CLOSE_CURSOR(id_cursor);
    DBMS_OUTPUT.PUT_LINE('Filas introducidas: ' || v_filas);
EXCEPTION
    WHEN OTHERS THEN
      DBMS_SQL.CLOSE_CURSOR(id_cursor);
      RAISE;
END introducir_fila;
/
```

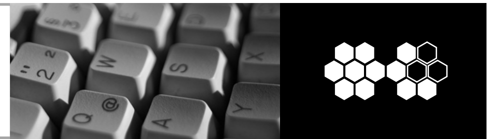
La línea `INSERT INTO mitabla VALUES (:val_1)` corresponde a la visualización del comando construido, que se incluye con fines didácticos.

En este ejemplo podemos apreciar que, para introducir dinámicamente comandos de manipulación de datos, utilizamos los procedimientos y funciones estudiados en el apartado anterior: OPEN_CURSOR, PARSE y CLOSE CURSOR.

Pero, además, utilizaremos el procedimiento EXECUTE (necesario para comandos de manipulación y consulta). También se utiliza en este caso el procedimiento BIND_VARIABLE para acoplar la variable con el valor de entrada.

A continuación, podemos observar un ejemplo de ejecución del procedimiento cuyo resultado es insertar una fila en la tabla `mitabla` que suponemos creada previamente:

```
SQL> execute introducir_fila('mitabla', 'HOLA MUNDO');
INSERT INTO mitabla VALUES (:val_1)
Filas introducidas: 1
Procedimiento PL/SQL terminado con éxito.
```



Podemos comprobar el resultado de la ejecución introduciendo la siguiente consulta:

```
SQL> select * from mitabla;
COL1
-------------------------------------------------------
HOLA MUNDO
```

Consultas con DBMS_SQL

El siguiente ejemplo permite obtener el número de empleado, el apellido y el oficio de los empleados que cumplan la condición que se especificará en la llamada:

```
CREATE OR REPLACE PROCEDURE consultar_emple
    (condicion VARCHAR2,
     valor VARCHAR2)
AS
    id_cursor INTEGER;
    v_comando VARCHAR2(2000);
    v_dummy      NUMBER;
    v_emp_no   emple.emp_no%TYPE;
    v_apellido emple.apellido%TYPE;
    v_oficio   emple.oficio%TYPE;
BEGIN
    id_cursor :=       DBMS_SQL.OPEN_CURSOR;
    v_comando := '     SELECT emp_no, apellido, oficio
                       FROM emple
                       WHERE ' || condicion  || ':val_1';

    DBMS_OUTPUT.PUT_LINE(v_comando);
    DBMS_SQL.PARSE(id_cursor, v_comando, DBMS_SQL.NATIVE);
    DBMS_SQL.BIND_VARIABLE(id_cursor, ':val_1', valor);

/* Se especifican las variables que recibirán los valores
de la selección*/

    DBMS_SQL.DEFINE_COLUMN(id_cursor, 1, v_emp_no);
    DBMS_SQL.DEFINE_COLUMN(id_cursor, 2, v_apellido, 14);
    DBMS_SQL.DEFINE_COLUMN(id_cursor, 3, v_oficio, 14);
    v_dummy := DBMS_SQL.EXECUTE(id_cursor);

/* La función FETCH_ROWS recupera filas y retorna el
número de filas que quedan */

    WHILE DBMS_SQL.FETCH_ROWS(id_cursor)>0 LOOP

/* A continuación se depositarán los valores recuperados
en las variables PL/SQL */
```

```
        DBMS_SQL.COLUMN_VALUE(id_cursor, 1, v_emp_no);
        DBMS_SQL.COLUMN_VALUE(id_cursor, 2, v_apellido);
        DBMS_SQL.COLUMN_VALUE(id_cursor, 3, v_oficio);
        DBMS_OUTPUT.PUT_LINE(v_emp_no || '*' || v_apellido
                        || '*' || v_oficio);
    END LOOP;
    DBMS_SQL.CLOSE_CURSOR(id_cursor);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_SQL.CLOSE_CURSOR(id_cursor);
        RAISE;
END consultar_emple;
/
```

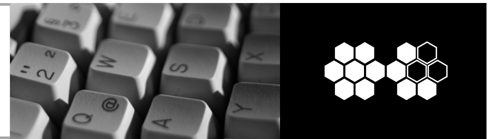
La principal diferencia de este programa con los anteriores es que, en este caso, como ocurre en todas las consultas, hay que recoger los valores obtenidos por la consulta. Para ello, se utilizan los siguientes procedimientos y funciones:

El paquete DBMS_SQL dispone de otras posibilidades que quedan fuera del objetivo de este libro. Para profundizar más en este tema, se recomienda consultar el manual de Oracle ***Application Developer's Guide.***

- DEFINE_COLUMN: sirve para especificar las variables que recibirán los valores de la selección. Se utilizará una llamada para cada variable indicando:
  - El identificador del cursor.
  - El número de la columna de la cláusula SELECT de la que recibirá el valor.
  - El nombre de la variable.
  - Si la variable es de tipo CHAR, VARCHAR2 o RAW, hay que especificar la longitud.
- FETCH_ROWS: es similar al comando FETCH de PL/SQL estático, pero con dos particularidades muy importantes:
  1. Devuelve un valor numérico que indica el número de filas que quedan por recuperar en el cursor. Puede servir, como en el ejemplo, para controlar el número de iteraciones del bucle.
  2. Para introducir los datos recuperados en las variables PL/SQL todavía hay que utilizar el siguiente procedimiento.
- COLUMN_VALUE: deposita el valor de un elemento del cursor (recuperado con FETCH_ROWS) cuya posición se especifica en una variable PL/SQL. Normalmente se trata de la variable indicada en COLUMN_VALUE para el mismo elemento del cursor.

La función EXECUTE con instrucciones DML devuelve el número de filas afectadas. Sin embargo, en el caso de las consultas (y también en órdenes DDL si se usa) el valor que devuelve es indefinido.

```
SQL>  EXECUTE CONSULTAR_EMPLE('SALARIO > ', 3850)
SELECT emp_no, apellido, oficio
FROM emple
WHERE SALARIO > :val_1
```



```
7566*JIMENEZ*DIRECTOR
7788*GIL*ANALISTA
7839*REY*PRESIDENTE
7902*FERNÁNDEZ*ANALISTA
```

B. SQL dinámico con NDS

Las versiones más recientes de Oracle permiten trabajar con NDS (*Native Dynamic SQL*) que tiene importantes ventajas respecto al uso del paquete DBMS_SQL:

- La sintaxis es más sencilla y parecida a SQL.
- Manejo directo de FETCH desde PL/SQL.
- Soporta objetos y tipos definidos por el usuario.
- Los programas se ejecutan más rápido.

Una de las principales novedades de NDS es el comando EXECUTE IMMEDIATE que recibe una cadena de tipo VARCHAR2 conteniendo el comando a ejecutar, lo analiza y ejecuta.

```
CREATE OR REPLACE PROCEDURE ejsqldin_nds
    (instruccion VARCHAR2)
AS
BEGIN
    EXECUTE IMMEDIATE instruccion;
END ejsqldin_nds;
/
```

La instrucción se puede recibir en uno o en varios parámetros, completa o incompleta. El comando se formará concatenando los parámetros, variables, literales, etcétera.

Por ejemplo, el procedimiento de abajo recibirá el nombre de una tabla, el de una columna de dicha tabla y un valor de dicha columna, y eliminará la fila o filas cuyo valor en la columna especificada coincida con el indicado.

```
CREATE OR REPLACE PROCEDURE borrar
    (nombre_tabla VARCHAR2,
     columnaref VARCHAR2,
     valor      VARCHAR2)
AS
     instruccion_borrado VARCHAR2(2000);
BEGIN
 instruccion_borrado := 'DELETE FROM ' || nombre_tabla ||
' WHERE ' || columnaref || ' = ' || valor;
    EXECUTE IMMEDIATE instruccion_borrado;
END;
```

Para probarlo podemos escribir:

```
SQL> EXECUTE BORRAR('EMPLE', 'EMP_NO', 7902);
```

También se pueden usar *variables de transferencia*. En este caso se indicará mediante opción USING las variables (o valores) que se asignaran a las variables de transferencia contenidas en la instrucción.

```
EXECUTE IMMEDIATE instruccion USING listadeparámetros;
```

La asignación de los parámetros o valores de USING a las variables de transferencia es posicional.

Por ejemplo, el procedimiento anterior usando una variable de transferencia para indicar el valor que queremos borrar (y evitar problemas con valores de tipos distintos al numérico) sería:

```
CREATE OR REPLACE PROCEDURE borrar
    (nombre_tabla VARCHAR2,
     columnaref VARCHAR2,
     valor      VARCHAR2)
AS
     instruccion_borrado VARCHAR2(2000);
BEGIN
  instruccion_borrado := 'DELETE FROM ' || nombre_tabla
|| ' WHERE ' || columnaref || ' = :vt1';  -- vt1 es la
variable de transferencia.
    EXECUTE IMMEDIATE instruccion_borrado
      USING valor;
END;
```

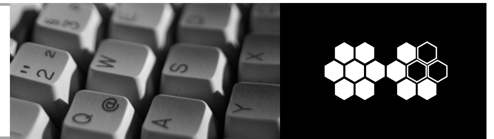
11.6 Objetos con PL/SQL

Oracle incluye la posibilidad de trabajar con objetos desde la versión 8. En sucesivas versiones se ha implementado soporte para características avanzadas propias de la programación orientada a objetos mediante PL/SQL (herencia, polimorfismo, etcétera).

A. El tipo OBJECT

El **tipo OBJECT** es una definición o molde de objeto. Es el equivalente a una clase en Java. Al definir un tipo OBJECT podemos definir atributos y métodos:

- Los **atributos** son las características que sirven para describir físicamente el objeto.
- Los **métodos** son las acciones que se pueden realizar con el objeto (procedimientos y funciones).



```
CREATE OR REPLACE
TYPE nombre_de_tipo AS OBJECT (lista_atributos, [lista_
métodos]);
```

Por ejemplo, podemos definir el tipo COCHE_OBJ con los atributos *marca*, *modelo*, *nbastidor*, *fechaf*, *valorff*, *matrícula*:

```
CREATE OR REPLACE TYPE coche_obj AS OBJECT (
    marca      VARCHAR2(20),
    modelo     VARCHAR2(10),
    nbastidor  NUMBER(15),
    fechaf     DATE,
    valorff    NUMBER(8,2),
    matricula  VARCHAR2(10)
);
```

El sistema responderá: `Tipo creado`.

Observamos que en este caso sólo hemos incluido atributos dentro de la declaración, pues no es obligatorio definir ningún método al crear un tipo OBJECT. En cambio, sí es obligatorio definir al menos un atributo. Además, los atributos deben cumplir:

- Se puede usar cualquier tipo soportado por la base de datos excepto ROWID, UROWID, LONG, LONG RAW, NCHAR, NVARCHAR2, NCLOB.
- No se pueden usar tipos específicos de PL/SQL no soportados por la base de datos.
- No se pueden incluir calificadores NOT NULL ni DEFAULT.
- Tampoco podemos usar el atributo %TYPE o %ROWTYPE para definir el tipo.
- Se pueden incluir tipos definidos como OBJECT dentro de la definición de otros. A estos se les llama **tipos OBJECT complejos.**

En caso de que haya algún error de compilación al crear el objeto aparecerá:

`… Warning: Type created with compilation errors.`

En estos casos, usaremos SHOW ERRORS y depuraremos los errores.

Una vez declarado el tipo podemos usarlo para declarar e inicializar objetos. Esto lo haremos en la sección declarativa igual que si se tratase de cualquier otro tipo predefinido:

```
DECLARE
…
 v_coche COCHE_OBJ := COCHE_OBJ('SEAT', 'LEON TDI', NULL,
                             NULL, NULL, NULL);
…
```

Observamos que al declarar el objeto lo hemos inicializado pues todo objeto debe ser inicializado antes de usarlo. Para ello, se utiliza el constructor por defecto del tipo, seguido de la lista de valores para los atributos (entre paréntesis).

B. Métodos

Normalmente, cuando creamos un objeto también crearemos los métodos que definen el comportamiento de ese objeto y que permiten actuar sobre él.

Los métodos son procedimientos y funciones que se especificarán después de los atributos del objeto indicando también el tipo de método, que puede ser:

- **MEMBER.** Son métodos que sirven para interactuar con los objetos. Se trata de funciones y procedimientos con un comportamiento y formato similares a los estudiados en los paquetes.

  En nuestro caso podríamos crear un método que se llame MATRICULAR que consiste en una función que asigna una matrícula a un objeto (una variable) de tipo COCHE_OBJ:

  ```
  ...
  MEMBER PROCEDURE matricular (mat VARCHAR2(10))
  ```

- **STATIC.** Son métodos estáticos independientes de las instancias del objeto. Existe una similitud en su comportamiento con las variables estáticas definidas en la cabecera de los paquetes dentro de la sección ejecutable opcional.

- **CONSTRUCTOR.** Sirve para inicializar un objeto. Se trata de una función cuyos argumentos son los valores de los atributos del objeto y que devuelve el objeto inicializado.

Para cada objeto existe un constructor predefinido por Oracle (cuyo nombre y atributos son los mismos que los del objeto).

Por tanto, no es necesario crear un constructor, pues disponemos de uno por defecto. No obstante, podemos sobrescribirlo y/o crear otros constructores adicionales; además, creando nuestros propios constructores podemos incluir valores por defecto, restricciones, etcétera.

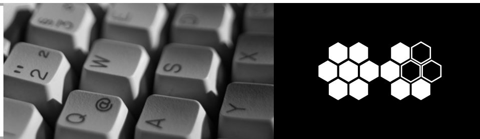
Los constructores llevarán en la cláusula `return` la expresión RETURN SELF AS RESULT.

Cabe subrayar que los métodos STATIC y MEMBER pueden ser procedimientos y funciones, mientras que los constructores CONSTRUCTOR son siempre funciones.

Una vez declarada la especificación de los métodos crearemos el cuerpo del nuevo tipo OBJECT (de manera similar a como lo hacíamos con el cuerpo de un paquete) mediante la instrucción:

```
CREATE OR REPLACE TYPE BODY nombre_de_tipo AS
<implementación de los métodos>
```

Donde *<implementación de los métodos>* incluye el código de todos los métodos ademas del tipo de los mismos según los formatos:



Para los procedimientos	Para las funciones
[STATIC | MEMBER] PROCEDURE	[STATIC | MEMBER | CONSTRUCTOR] FUNCTION
nombre_procedimiento	nombre_funcion
[(lista_de_parámetros)]	[(lista_de_parámetros)]
IS	RETURN tipo_valor_retorno
declaraciones;	IS
…	declaraciones;
BEGIN	…
instrucciones;	BEGIN
…	instrucciones;
END;	…
	END;

En nuestro ejemplo, podemos añadir a la declaración del tipo COCHE_OBJ los siguientes métodos:

- `mostrar_coche`: visualiza todos los datos del coche.
- `cambiar_precio_pct`: modifica el precio aplicando el porcentaje de subida o bajada.
- `ver_precio_total`: devuelve el precio con IVA.

La especificación de COCHE_OBJ con los tres métodos enunciados será:

```
CREATE OR REPLACE TYPE coche_obj AS OBJECT (
    marca      VARCHAR2(20),
    modelo     VARCHAR2(10),
    nbastidor  NUMBER(15),
    fechaf     DATE,
    valorff    NUMBER(8,2),
    matricula  VARCHAR2(10),
MEMBER PROCEDURE mostrar_coche,
MEMBER PROCEDURE cambiar_precio_pct (pct NUMBER),
MEMBER FUNCTION ver_precio_total RETURN NUMBER
);
```

Una vez creada la especificación crearemos el cuerpo:

```
CREATE OR REPLACE TYPE BODY coche_obj
AS
-------------------------------
MEMBER PROCEDURE mostrar_coche
  IS
  BEGIN
      DBMS_OUTPUT.PUT_LINE(marca || ' ' || modelo || ' ' ||
            nbastidor || ' ' || fechaf || ' ' ||
            valorff || ' ' || matricula);
  END;
```

Para referirnos a la instancia actual del objeto podemos usar la palabra reservada ***SELF*** (igual que en otros lenguajes OO).

```
-----------------------------------------------
MEMBER PROCEDURE cambiar_precio_pct(pct NUMBER)
  IS
  BEGIN

      SELF.valorff := SELF.valorff * (1 + (pct/100));
  END;
---------------------------------
MEMBER FUNCTION ver_precio_total
  RETURN NUMBER
  IS
    iva CONSTANT NUMBER(2,2) := 0.16;
    precio_total NUMBER(10,2);
  BEGIN
    precio_total := SELF.valorff * (1 + iva);
    RETURN precio_total;
  END;
---------------------------------
END;
```

Ahora podemos usar el tipo COCHE_OBJ para crear intancias o variables de ese tipo que aceptarán los métodos implementados:

```
SET SERVEROUTPUT ON
DECLARE

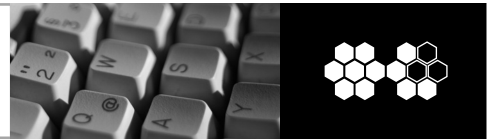
  /* declaramos y creamos una instancia. El identificador
se usa dos veces la primera se refiere al tipo, la
segunda (en negrita) invoca al constructor del tipo con
los valores se especifican entre paréntesis */

  v_coche COCHE_OBJ := COCHE_OBJ('SEAT', 'LEON TDI',
                       123456767,SYSDATE, 20000, NULL);
BEGIN

  /* invocaremos los métodos disponibles para el objeto
mediante nombreinstancia.nombremetodo */
  v_coche.mostrar_coche;
  v_coche.cambiar_precio_pct(+10);
  v_coche.matricula := '5678-EZX';
  DBMS_OUTPUT.PUT_LINE('Nuevo precio incrementado en 10%
    con IVA:' || v_coche.ver_precio_total);
  v_coche.mostrar_coche;
END;
```

El resultado de la ejecución será:

```
SEAT LEON TDI 123456767 22/09/05 20000
Nuevo precio incrementado en 10% con IVA:25520
SEAT LEON TDI 123456767 22/09/05 22000 5678-EZX
--- Fin 170 mm ---
```



C. Ordenaciones y comparaciones con objetos

En muchas ocasiones necesitaremos poder comparar, e incluso ordenar, datos de tipos definidos como OBJECT. Este problema aparece especialmente cuando se usan operadores relacionales o cláusulas SQL como GROUP BY, ORDER BY, DISTINCT, etcétera, pues no hay un criterio de ordenación o comparación predefinido para estos datos, por tanto, normalmente deberemos crearlo mediante *métodos MAP*.

Los **métodos MAP** consisten en una función que devuelve un valor de tipo escalar (CHAR, VARCHAR2, NUMBER, DATE, ...) que será el que se utilice en las comparaciones y ordenaciones aplicando los criterios preestablecidos para estos tipos de datos.

En nuestro ejemplo, podemos establecer que los valores a partir de los cuales se ordenará serán la matrícula (VARCHAR2). También podríamos haber optado por el número de bastidor, o por el precio, o por una combinación de atributos como marca||modelo, etcétera. Una vez decidido el criterio incluiremos el método en el objeto usando el especificador MAP:

– Primero en la especificación del tipo indicaremos el nombre y el tipo de valor devuelto:

```
MAP MEMBER FUNCTION por_matricula RETURN CHAR
```

– Después en el cuerpo de la definición del tipo escribiremos el código correspondiente a la función:

```
MAP MEMBER FUNCTION por_matricula RETURN CHAR
IS
BEGIN
    RETURN SELF.matricula;
END;
```

En lugar de MAP pueden usarse métodos ORDER, que no entraremos a detallar pues suelen ser menos funcionales y eficientes.

Conceptos básicos



- **Los disparadores de base de datos** son programas PL/SQL que se disparan cuando se producen ciertos eventos:
  - **Disparadores de tablas:** asociados a una tabla. Se disparan cuando se produce un evento de manipulación que afecta a la tabla (inserción, borrado o modificación de filas).
  - **Disparadores de sustitución:** asociados a vistas. Se disparan cuando se intenta ejecutar un comando de manipulación que afecta a la vista (inserción, borrado o modificación de filas).
  - **Disparadores del sistema:** se disparan cuando ocurre un evento del sistema o una instrucción de definición de datos (creación, modificación o eliminación de una tabla u otro objeto).

- **Los registros** son estructuras compuestas de otras más simples (campos) que pueden ser de distintos tipos. Se crean:
  1. Se define el tipo genérico del registro: `TYPE nombre_tipo IS RECORD (definición_de_campos);`
  2. Se declaran las variables que se necesiten de ese tipo: `nombre_variable   nombre_tipo;`

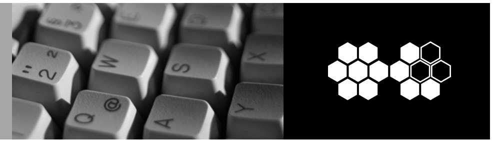
- **Los colecciones** son estructuras compuestas por listas de elementos. Pueden ser de tres tipos:
  - ***Varrays:*** índice secuencial, longitud fija, acceso desde SQL y PL/SQL, soportados en tablas de la base de datos.
  - **Tablas anidadas:** igual que los *varrays* pero se pueden inicializar elementos adicionales dinámicamente.
  - **Tablas indexadas o *arrays* asociativos:** el índice no es secuencial, todos los elementos se crean dinámicamente, no requieren inicialización, no soportadas en tablas ni en SQL, no son objetos.

- **Los paquetes** se utilizan para agrupar y guardar programas y otros objetos en la base de datos. Hay dos elementos:
  - **Especificación:** contiene declaraciones públicas de subprogramas, tipos, constantes, variables, cursores, excepciones, etcétera. Los objetos declarados en la especificación son accesibles también desde fuera del paquete.
  - **Cuerpo:** contiene los detalles de implementación de todos los objetos del paquete (el código de los programas, etcétera). También puede incluir declaraciones de objetos accesibles solamente desde los objetos del paquete.

  Los objetos declarados o definidos en un paquete se pueden usar:
  - Desde el mismo paquete por los demás objetos del mismo, con independencia de que hayan sido o no declarados en la especificación. No necesitan utilizar la notación de punto para referirse a los objetos del mismo paquete.
  - Desde fuera del paquete solamente si han sido declarados en la especificación. Se usará la notación de punto.

- **SQL dinámico** permite ejecutar instrucciones resolviendo referencias a objetos en el momento de la ejecución. Hay dos formas de trabajar con SQL dinámico:
  - El paquete DBMS_SQL.
  - SQL dinámico nativo (NDS), disponible sólo a partir de la versión 9i.

- El **tipo OBJECT** es una definición o molde de objeto. En un tipo OBJECT podemos definir atributos y métodos:
  - Los **atributos** son las características que sirven para describir físicamente el objeto.
  - Los **métodos** son las acciones que se pueden realizar con el objeto (procedimientos y funciones).



Actividades complementarias



1 Escribe un disparador de base de datos que permita auditar las operaciones de inserción o borrado de datos que se realicen en la tabla EMPLE según las siguientes especificaciones:

- Se creará desde SQL*Plus la tabla *auditaremple* con la columna *col1 VARCHAR2(200)*.
- Cuando se produzca cualquier manipulación, se insertará una fila en dicha tabla que contendrá: fecha y hora, número de empleado, apellido y la operación de actualización INSERCIÓN o BORRADO.

2 Escribe un *trigger* que permita auditar las modificaciones en la tabla EMPLEADOS, insertando los siguientes datos en la tabla *auditaremple:* fecha y hora, número de empleado, apellido, la operación de actualización MODIFICACIÓN y el valor anterior y el valor nuevo de cada columna modificada (sólo en las columnas modificadas).

3 Suponiendo que disponemos de la vista:

```
CREATE VIEW DEPARTAM AS
SELECT DEPART.DEPT_NO, DNOMBRE, LOC,
COUNT(EMP_NO) TOT_EMPLE
FROM EMPLE, DEPART
WHERE EMPLE.DEPT_NO (+) =
DEPART.DEPT_NO
GROUP BY DEPART.DEPT_NO, DNOMBRE, LOC;
```

Construye un disparador que permita realizar actualizaciones en la tabla *depart* a partir de la vista *departam,* de forma similar al ejemplo del *trigger t_ges_emplead*. Se contemplarán las siguientes operaciones:

- Insertar y borrar departamento.
- Modificar la localidad de un departamento.

4 Escribe un paquete para gestionar los departamentos. Se llamará *gest_depart* e incluirá, al menos, los siguientes subprogramas:

- *insertar_nuevo_depart:* inserta un departamento nuevo. Recibe el nombre y la localidad del nuevo departamento. Creará el nuevo departamento comprobando que el nombre no se duplique y le asignará como número de departamento la decena siguiente al último número de departamento utilizado.
- ***borrar_depart:*** borra un departamento. Recibirá dos números de departamento: el primero corresponde al departamento que queremos borrar y el segundo, al departamento al que pasarán los empleados del departamento que se va a eliminar. El procedimiento se encargará de realizar los cambios oportunos en los números de departamento de los empleados correspondientes.
- ***modificar_loc_depart:*** modifica la localidad del departamento. Recibirá el número del departamento que se modifica y la nueva localidad, y realizará el cambio solicitado.
- ***visualizar_datos_depart:*** visualizará los datos de un departamento cuyo número se pasará en la llamada. Además de los datos relativos al departamento, se visualizará el número de empleados que pertenecen actualmente al departamento.
- ***visualizar_datos_depart:*** versión sobrecargada del procedimiento anterior que, en lugar del número del departamento, recibirá el nombre del departamento. Realizará una llamada a la función *buscar_depart_por_nombre* que se indica en el apartado siguiente.
- ***buscar_depart_por_nombre:*** función local al paquete. Recibe el nombre de un departamento y devuelve el número del mismo.

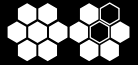
5 Crea un procedimiento que permita consultar todos los datos de la tabla *depart* a partir de una condición que se indicará en la llamada al procedimiento.

Administración de Oracle I

12

En esta unidad aprenderás a:

1. Entender los componentes de una base de datos de Oracle.
2. Comprender la estructura de la memoria y los procesos de Oracle.
3. Instalar una base de datos Oracle.
4. Crear y configurar una base de datos Oracle.
5. Crear, modificar y borrar usuarios.
6. Consultar las vistas con información del sistema.
7. Establecer y retirar privilegios.



12.1 Introducción

La tarea de administración de una base de datos para varios usuarios es bastante compleja, por lo que se suele encomendar a una o varias personas. El administrador de la base de datos (DBA: *Data Base Administrator*) suele ser un usuario muy experimentado capaz de enfrentarse a los problemas de los demás usuarios y a los que el sistema va planteando. Son tareas del administrador de Oracle: instalar Oracle, diseñar y crear una base de datos, arrancar y detener la base de datos, crear y controlar usuarios, conceder privilegios, gestionar el espacio, hacer copias de seguridad y recuperar la base de datos.

12.2 ¿Qué es Oracle 10g?

Oracle 10g es la nueva suite de productos software de la compañía Oracle, basados todos ellos en la tecnología *Grid Computing* (o computación Grid). El Grid es una nueva arquitectura que agrupa múltiples servidores y recursos de almacenamiento y procesamiento en una estructura más económica y flexible que atiende a todas las necesidades de la organización y donde los recursos para el procesamiento de datos están disponibles para los usuarios según los vayan necesitando.

Los tres elementos sobre los que se articula *Oracle Grid Computing* son: *Oracle Database 10g*, *Oracle Application Server 10g* y *Oracle Enterprise Manager 10g*:

- **Oracle Database 10g.** Es el motor de la base de datos. Dispone de herramientas capaces de gestionar eficazmente el almacenamiento de la información, utilizar de forma óptima los recursos, ofrecer un máximo nivel de atención en los servicios ofrecidos, etc. Gestiona de forma eficaz toda la información relacional, email, documentos, multimedia, XML y espacial.

- **Oracle Application Server 10g.** Es el servidor de aplicaciones Oracle. Como nivel intermedio de una arquitectura de tres niveles consiste en un conjunto de servicios que permite a las aplicaciones basadas en exploradores web interactuar con las bases de datos de Oracle.

- **Oracle Enterprise Manager 10g.** Proporciona un marco adecuado para llevar a cabo las tareas de administración de la base de datos. Se accede a través del navegador web mediante la URL: http://nombre_del_host:5500/em. Se obtiene una pantalla desde la que hay que escribir el nombre de usuario, la clave y el rol administrativo. Al instalar Oracle Database 10g, se instala este elemento.

12.3 Arquitectura Oracle

Para comprender mejor una base de datos necesitamos conocer su arquitectura. Cuando hablamos de una «base de datos» no sólo nos estamos refiriendo a los datos físicos, sino también a la combinación de objetos físicos, de memoria y de proceso que se describen a continuación.

A. Componentes de la base de datos

Los componentes de una base de datos Oracle son: archivos de datos (*database files*), archivos de diario o de transacciones (*log files*) y archivos de control (*control files*).

Archivos de datos

Contienen toda la información de la base de datos: datos de usuario y datos de sistema. Antes de introducir datos en la base de datos, es necesario crear un espacio para las tablas (*tablespace*) y después crear una tabla, dentro de ese espacio, en la que introducir los datos. Los *tablespaces* nos ayudan a organizar la información contenida en la base de datos; así, podemos tener un *tablespace* para almacenar los datos de la aplicación de almacén, otro para almacenar los datos de la aplicación de nóminas, etcétera.

Cada *tablespace* consta de uno o más archivos en disco. Un archivo de datos sólo puede pertenecer a un único *tablespace*. Al instalar Oracle se crean varios *tablespaces*, algunos son:

- **SYSTEM.** En él se almacena toda la información que Oracle necesita para gestionarse a sí misma, por ejemplo: el diccionario de datos. Se almacena en el archivo SYSTEM01.DBF. En la versión 10g existe el *tablespace* SYSAUX, que es un espacio de tablas auxiliar a SYSTEM, y se crea automáticamente. Su misión es descargar de trabajo a SYSTEM.

- **USERS.** Contiene información personal de los usuarios. Normalmente, es el lugar en el que el DBA nos deja almacenar las tablas para realizar pruebas. Se almacena en el archivo USERS01.DBF.

- **TEMP.** Aquí Oracle almacena las tablas temporales (para gestionar sus transacciones). Se almacena en el archivo TEMP01.DBF.

- **UNDOTBS1.** En él es donde Oracle guarda la información de deshacer. Se utiliza para almacenar la imagen anterior de los datos antes de permitir actualizaciones. Esto permite recuperar los datos cuando no se completa una transacción. Se almacena en el archivo UNDOTBS01.DBF. Este *tablespace* contiene los segmentos de Rollback. Sin éstos no se podrían realizar transacciones. Cuando se realiza una transacción se asigna un segmento por defecto. La siguiente consulta visualiza los segmentos de Rollback, su propietario y el *tablespace* donde está:

```
SQL> SELECT SEGMENT_NAME,OWNER,TABLESPACE_NAME FROM
DBA_ ROLLBACK_SEGS;
 SEGMENT_NAME          OWNER       TABLESPACE_NAME
-------------------- ---------- ---------------------
    SYSTEM             SYS              SYSTEM
   _SYSSMU1$           PUBLIC          UNDOTBS1
   _SYSSMU2$           PUBLIC          UNDOTBS1
   _SYSSMU3$           PUBLIC          UNDOTBS1
   _SYSSMU4$           PUBLIC          UNDOTBS1
   _SYSSMU5$           PUBLIC          UNDOTBS1
   _SYSSMU6$           PUBLIC          UNDOTBS1
   _SYSSMU7$           PUBLIC          UNDOTBS1
```



```
 _SYSSMU8$            PUBLIC          UNDOTBS1
 _SYSSMU9$            PUBLIC          UNDOTBS1
_SYSSMU10$            PUBLIC          UNDOTBS1
_SYSSMU11$            PUBLIC          UNDOTBS1
```

Registros de rehacer o Redo Log: el registro de las transacciones

Se trata de archivos de datos en los que Oracle registra todos los cambios que se efectúan sobre los datos (INSERT, UPDATE y DELETE) de la base de datos dentro de la caché de buffers de la base de datos. Estos archivos se utilizan en situaciones de fallo para recuperar datos validados que no se han escrito en los archivos de datos. Suele haber varios registros de rehacer y conviene almacenarlos en discos diferentes para evitar pérdidas debido a fallos en el disco. Normalmente se crean varios grupos de Redo Log online, formado cada uno de ellos por varios miembros, los miembros de un grupo contienen copias idénticas. El proceso en segundo plano LGWR escribe simultáneamente la misma información en todos los archivos de Redo Log de un grupo. Oracle crea un mínimo de dos grupos de Redo Log online, los miembros se llaman REDO01.LOG, REDO02.LOG y REDO03.LOG.

Un registro de Redo Log contiene: identificación de la transacción, dirección de bloque, número de fila, número de columna y valor anterior y nuevo del dato modificado.

Cuando se está utilizando un grupo de archivos de Redo Log el servidor Oracle le asigna un número de secuencia para identificarlo. Este número de secuencia se almacena en el archivo de control y en la cabecera de todos los archivos de datos. Esto se hace para en situaciones de fallo recuperar los datos a partir de un número de Redo Log determinado.

Archivos de control

Contienen información sobre los archivos asociados con una base de datos Oracle. Todas las modificaciones importantes que se hagan en la estructura de la base de datos se registran en el archivo de control. Estos archivos mantienen la integridad de la base de datos. Oracle recomienda que la base de datos tenga un mínimo de dos archivos de control en discos diferentes. Si se daña un archivo de control debido a un fallo en disco, se podría restaurar utilizando la copia intacta del archivo de control del otro disco. La información del archivo de control sólo la puede modificar el servidor Oracle. Si se dañan los archivos la base de datos no funcionará correctamente. Es aconsejable realizar copias de seguridad de estos archivos cada vez que se produzca un cambio en la estructura física de la base de datos. Los archivos de control que se crean en la instalación son: CONTROL01.CTL, CONTROL02.CTL y CONTROL03.CTL y contienen la siguiente información:

- El nombre y el identificador de la base de datos.
- Registro de la hora y fecha de creación de la base de datos.
- Los nombres y ubicaciones de los archivos de datos asociados y los archivos de Redo Logs.
- El historial de los Redo Log, que se registra durante los cambios de logs.

- La ubicación y el estado de los Redo Logs archivados.
- La ubicación y el estado de las copias de seguridad.
- Número de secuencia de log actual, que se registra cuando se producen los cambios de log.
- Información sobre *checkpoints* (puntos de control que se dan cuando se llena el Redo Log, cuando se detiene la base de datos, etcétera).
- Estado *on-line* y *off-line* de los archivos de datos.

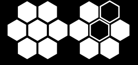
Para obtener información del archivo de control se pueden consultar las siguientes vistas a las que sólo tienen autorización los usuarios administradores (aquellos con el rol DBA), SYS y SYSTEM, estos dos últimos tienen que conectarse como SYSDBA desde el símbolo del sistema: `C:\>SQLPLUS SYS/ARM AS SYSDBA`.

ARM es la clave de SYS. Esta clave es la que se crea en la instalación. Si hay varias instancias en la base de datos pondremos: **`C:\>SQLPLUS SYS@ORCL/ARM AS SYSDBA`,** para conectarnos a ORCL.

Para ver los usuarios que tienen privilegios SYSDBA y SYSOPER debemos consultar la vista V$PWFILE_USERS. Las vistas para obtener información del archivo de control son:

- V$CONTROLFILE: visualiza nombre y ubicación de los archivos de control.
- V$CONTROLFILE_RECORD_SECTION: visualiza información sobre las diferentes secciones de los archivos de control, por ejemplo el número máximo de archivos.
- V$DATAFILE: visualiza información detallada de los archivos de datos.
- V$DATAFILE_HEADER: visualiza información de la cabecera del archivo de datos del archivo de control.
- V$THREAD: contiene información del *thread*, por ejemplo, acerca de los grupos de Redo Log.
- V$TEMPFILE: visualiza información de los archivos temporales.
- V$TABLESPACE: visualiza información de los espacios de tablas o *tablespaces*.
- V$DATABASE: visualiza información detallada de la base de datos.
- V$ROLLNAME Y V$ROLLSTAT: visualizan información acerca de los segmentos de Rollback.
- V$LOG, y V$LOGFILE: visualiza información de los archivos de Redo Log. Por ejemplo, si deseamos ver información de los grupos de Redo Logs y el estado actual tecleamos:

```
SQL> select group#, sequence#, bytes, members, status
from v$log;
```



GROUP#	SEQUENCE#	BYTES	MEMBERS	STATUS
1	185	10485760	1	CURRENT
2	183	10485760	1	INACTIVE
3	184	10485760	1	INACTIVE

B. Estructura de la memoria

Los procesos del usuario (cliente) y del servidor se comunican consigo mismos y entre ellos por medio de estructuras de memoria. Oracle utiliza dos tipos de estructuras de memoria: la SGA (*System Global Area*) y la PGA (*Program Global Area*).

Área global del sistema SGA

Es un grupo de estructuras de memoria que sirven para almacenar los datos de la base de datos que se han consultado más recientemente. Contiene información de datos y de control para el servidor Oracle. Se descompone en las siguientes zonas:

- **Conjunto Compartido o área SQL compartida:** formada por la *caché de diccionario de datos* y la *caché de biblioteca*. La **caché de diccionario de datos** contiene información acerca de las últimas definiciones utilizadas en la base de datos: archivos de bases de datos, columnas, usuarios, privilegios y otros objetos (por ejemplo, si un usuario puede acceder o no a una tabla). La **caché de biblioteca** contiene información sobre las instrucciones SQL ejecutadas sobre la base de datos: el texto de la sentencia, el código analizado y el plan de ejecución. La segunda vez que un usuario ejecuta una sentencia idéntica SQL ya ejecutada puede aprovecharse del análisis disponible en esta área para acelerar su ejecución.

- **Caché de Buffers:** contiene copias de los últimos bloques de datos leídos y utilizados de los archivos de datos. Los usuarios acceden a los datos en esta zona de la memoria. También se le conoce como buffer del bloque de datos o buffer de datos.

- **Conjunto Grande:** es un área de memoria opcional. Se utiliza para almacenar estructuras grandes de la memoria que no están directamente relacionadas con el procesamiento de sentencias SQL. Por ejemplo, los bloques de datos que se copian durante las operaciones de copia de seguridad y restauración. Se define cuando se utiliza la opción de servidor compartido o cuando se realizan con frecuencia operaciones de copias de seguridad y restauración.

- **Conjunto Java:** especifica el tamaño para satisfacer los requisitos de análisis de los comandos Java y almacena el código Java.

- **Buffer del registro de rehacer** (Redo Log buffer): en esta área se registran las transacciones (INSERT, UPDATE, DELETE) o cambios en la base de datos antes de escribirse en los archivos de registro de rehacer. Se utiliza por los procesos en segundo plano y servidor para hacer un seguimiento de los cambios realizados en la base de datos.

Área global del programa o de procesos PGA

Es la zona de memoria utilizada por un único proceso de usuario de Oracle, y contiene datos e información del proceso. La memoria de la PGA no se comparte. Cuando un usuario se conecta a la base de datos, por ejemplo, desde SQL Plus, se crea un proceso de usuario. Oracle comprueba y valida ese usuario e inmediatamente le asigna un proceso de servidor que lleva asociada su PGA. El proceso de servidor se comunica con la instancia Oracle en nombre del proceso de usuario que se ejecuta en el cliente, y ejecutará las sentencias SQL en nombre del usuario. La PGA contiene la información y los datos de un único proceso de servidor o de proceso de segundo plano. En un servidor dedicado la PGA incluye los siguientes componentes:

- Área de ordenación para las ordenaciones de sentencias SQL.
- Información de la sesión: privilegios de usuarios y estadísticas del rendimiento.
- Estado de cursor que indica la etapa en el procesamiento de las sentencias que la sesión utiliza en ese momento.

C. Procesos de soporte de la base de datos

Las relaciones entre las estructuras físicas y de memoria de la base de datos se mantienen y aplican mediante una serie de procesos soporte (de segundo plano, *background* o de fondo). El número de estos procesos varía en función de la configuración de la base de datos. La base de datos gestiona estos procesos y necesita poco trabajo administrativo.

Hay un conjunto de procesos del servidor que ayuda a la base de datos a funcionar: son los *procesos soporte* o *de segundo plano*. En la figura 12.1 se puede observar una visión general de una arquitectura Oracle.

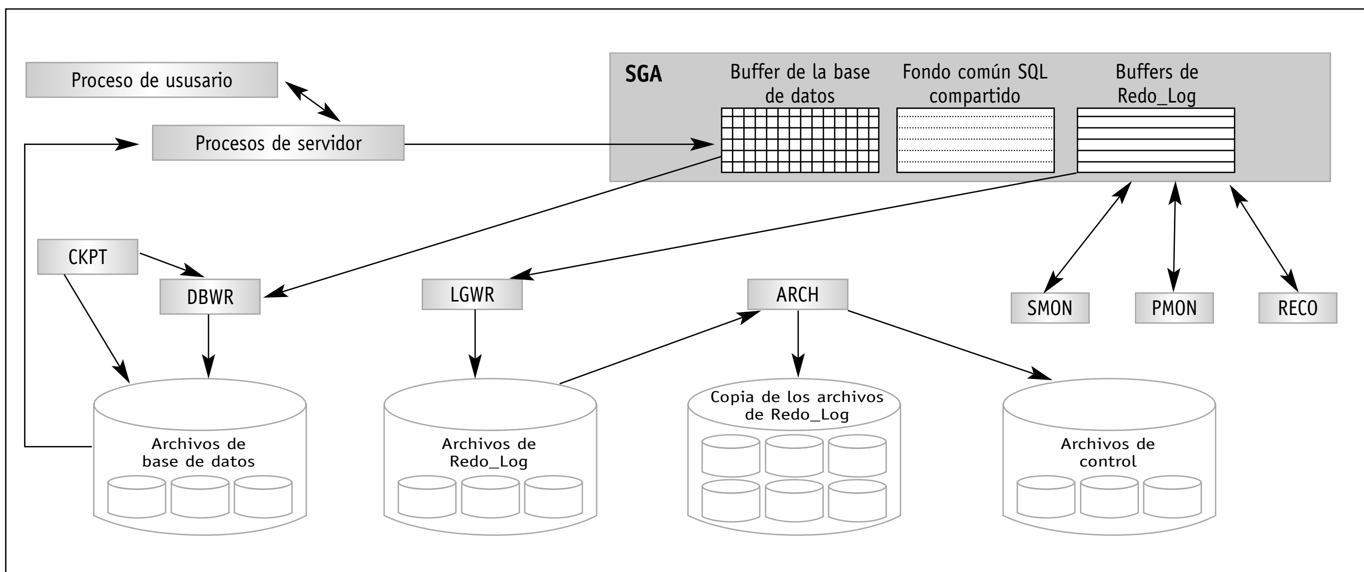
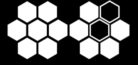


Figura 9.1. *Procesos y áreas de memoria Oracle.*



Los procesos en segundo plano son los siguientes:

- **Escritor de bases de datos DBWR** (*DataBase WRiter*). Este proceso es el responsable de gestionar el contenido del buffer de datos de la SGA. Lee los bloques de los archivos de datos, los almacena en la SGA y realiza escrituras de los bloques modificados en los archivos de datos y de Rollback. Cuando un usuario solicita una petición cuyos datos no están en el buffer de datos, el DBWR se encargará de llevar al buffer un nuevo bloque de información del archivo de datos. El DBWR escribe en los archivos de datos cuando: el número de bloques sucios (cualquier bloque de datos que se cambie en la caché de buffers de datos) alcanza el 90 % de ocupación, o cuando se produce un *timeout* (cada tres segundos), o cuando se produce un punto de control o *checkpoint*.

- **Punto de control o comprobación CKPT** (*ChecKPoinT*). Estos puntos provocan que el DBWR escriba en los archivos de datos todos los bloques del buffer de datos que se hayan modificado desde el último punto de control, y que actualice las cabeceras de los archivos de datos, de Redo Logs y de los archivos de control para registrar el punto de control y reflejar que se ha completado con éxito la escritura. El *checkpoint* es un medio de sincronizar la caché de buffer de base de datos con el archivo de datos. El número de punto de control actúa como marcador de sincronización, si los archivos de datos, de Redo Logs y de control tienen el mismo número de punto de control se considera que el estado de la base de datos es consistente. El archivo de control confirma en el inicio de la base de datos que todos los archivos están en el mismo punto de control. La no coincidencia en las cabeceras provocará un fallo y la base de datos no se podrá abrir, con lo que será necesario realizar una recuperación.

  Los *checkpoint* se producen de forma automática cuando se llena un archivo de registro de rehacer Redo Log, y cuando se cierra una instancia (SHUTDOWN) con la opción normal, transaccional o inmediata. También se puede forzar de forma manual por el administrador o utilizando parámetros de inicialización como LOG_CHECKPOINT_INTERVAL o LOG_CHECKPOINT_TIMEOUT.

  Los siguientes comandos fuerzan cambios de log y puntos de control:

  ```
  SQL> ALTER SYSTEM SWITCH LOGFILE;
  SQL> ALTER SYSTEM CHECKPOINT;
  ```

- **Escritor de registros LGWR** (*LoG WRiter*). Gestiona la escritura del contenido del buffer del registro de rehacer de la SGA a los archivos de Redo Log online. Es el único proceso que escribe en los archivos de registro de rehacer y el único que lee los buffers de este registro. Los registros de Redo se escriben en uno de los grupos, a los que el proceso LGWR denomina grupo Redo Log online actual, en las siguientes situaciones:

  – Cuando se valida una transacción.

  – Cuando el buffer Redo Log se llena a un tercio de su capacidad.

  – Cuando hay más de un mega de registros cambiados en el buffer de Redo Log.

El proceso LGWR escribe en los archivos de Redo Log de forma cíclica.

– Cuando se produce un *timeout* (cada tres segundos).

– Antes que el DBWR escriba los bloques modificados de la caché de buffers de la base de datos a los archivos de datos.

Esta operación permite que Oracle pueda recuperarse en cualquier momento si hay fallos. El proceso DBWR debe esperar al escritor de registros antes de escribir los bloques modificados desde los buffers del bloque de datos a los archivos de datos; es decir, en primer lugar se escribe la transacción en los registros de rehacer y luego se escribe en la base de datos.

LGWR escribe en los archivos de Redo Log de forma secuencial, cuando se llena un grupo, comienza a escribir en el siguiente, si se llena el último grupo volverá a escribir en el primero.

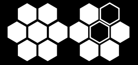
- **Supervisor del sistema SMON** (*System MONitor*). Comprueba la consistencia de la base de datos. El supervisor del sistema es un proceso obligatorio que se ocupa de todas las recuperaciones que sean precisas durante el arranque de la base de datos. La limpia eliminando datos de las transacciones que el sistema ya no necesita y compacta los huecos libres en los archivos de datos. Se activa de forma periódica para comprobar si es necesaria su intervención.

Cuando Oracle se ejecuta en modo ARCHIVELOG, la base de datos realiza una copia de los archivos de Redo Log antes de sobrescribirlos.

- **Supervisor de proceso PMON** (*Process MONitor*). Realiza una limpieza de recursos al terminar la ejecución de los procesos. Restaura las transacciones no validadas de los procesos de usuario que abortan, liberando los bloques y los recursos de la SGA. Cuando hay procesos fallidos el PMON deshace los cambios de la transacción actual de usuario, libera los bloqueos de tablas y filas actuales, y libera otros recursos que el usuario necesita en ese momento. Al igual que SMON, se activa de forma periódica para comprobar si es necesaria su intervención.

- **Archivador ARCH** (*ARCHiver*). Es opcional, y archiva en disco o cinta una copia de los Redo Log cuando están llenos para una posible recuperación por fallo de disco. Para que se produzca el archivado, la base de datos tiene que estar abierta en modo ARCHIVELOG, esto se decide al crear la base de datos. El proceso ARC0 se inicia haciendo una copia de seguridad del grupo de logs llenos en cada cambio de log. Archiva automáticamente los Redo Log online antes de que se puedan volver a utilizar, con el fin de proteger los cambios realizados en la base de datos. La copia automática se activa con el parámetro LOG_ARCHIVE_START = TRUE. Para obtener información acerca del archivado podemos ejecutar las siguientes sentencias:

```
SQL> ARCHIVE LOG LIST
Database log mode       No Archive Mode
Automatic archival       Disabled
Archive destination      USE_DB_RECOVERY_FILE_DEST
Oldest online log sequence   183
Current log sequence      185
SQL> SELECT NAME, LOG_MODE FROM V$DATABASE;
NAME   LOG_MODE
--------- ------------
ORCL   NOARCHIVELOG
```

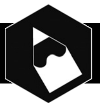


- **Recuperador RECO** (*RECOverer*). Es opcional. Recupera transacciones distribuidas dudosas; se usa en bases de datos Oracle distribuidas.

RESUMEN	
La base de datos Oracle más elemental consta de:	**Internamente, esta base de datos contiene:**
– Uno o más archivos de datos. – Uno o más archivos de control. – Dos o más archivos de Redo Log.	– Varios usuarios/esquemas. – Uno o más segmentos de Rollback. – Uno o más espacios de tablas. – Tablas del diccionario de datos. – Objetos de usuarios (tablas, vistas, etc.).
El servidor que accede a esta base de datos consta, como mínimo, de: – Un SGA (que contiene caché del buffer del bloque de datos, caché del buffer del registro de rehacer, el fondo común SQL). – El proceso SMON. – El proceso PMON. – El proceso DBWR. – El proceso LGWR. – Procesos de usuario con PGA asociado.	

Veamos ahora mediante un caso práctico qué hace Oracle cuando hacemos una consulta a la base de datos y cuando hacemos una actualización de una fila.

Caso práctico



1 **Se desea consultar la nota de la alumna «Luisa Martínez» en la asignatura de «Informática». Para ello, hacemos la siguiente consulta a la tabla NOTAS_ALUMNOS de la base de datos a través de la siguiente sentencia SQL:**

```
SELECT NOTA FROM NOTAS_ALUMNOS WHERE ASIGNATURA ='Informática' AND NOMBRE_ALUMNO='Luisa
Martínez';
```

La sentencia SQL es un proceso de usuario que sigue estos pasos:

- El proceso de usuario pasa la sentencia SQL al proceso del servidor que reserva la PGA correspondiente, y se comunica con la instancia.
- Los procesos del servidor buscan en el fondo común SQL una versión ejecutable de la sentencia. Si la encuentran, se ejecuta la sentencia SQL; si no se encuentra, el proceso servidor realiza el análisis siguiendo estos pasos:
  - Valida la sentencia comprobando la sintaxis.
  - Realiza búsquedas en el diccionario de datos para validar las definiciones de tabla y columna.

(Continúa)

(Continuación)

- Establece bloqueos de análisis en objetos de forma que no cambien sus definiciones durante el análisis de la sentencia.
- Comprueba los privilegios que el usuario dispone para acceder a los objetos.
- Determina el plan de ejecución de la sentencia.
- Carga la sentencia y el plan de ejecución en el área o fondo SQL compartido.

• Si los datos requeridos no están en el buffer del bloque de datos, por medio del proceso **DBWR** se leen los datos de los archivos de datos y se colocan en los buffers de datos de la SGA.

• Una vez que los datos se encuentran en los buffers de datos, el proceso de servidor los envía al proceso de usuario y el usuario puede leer la nota.

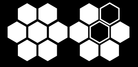
Éste es el ejemplo más sencillo, pues no se ha hecho ninguna modificación de los datos.

• Suponemos que cambiamos la nota de Informática de Luisa Martínez. Se modifica la columna NOTA de la tabla NOTAS_ALUMNOS con el valor 7:

```
UPDATE NOTAS_ALUMNOS SET NOTA = 7 WHERE ASIGNATURA ='Informática' AND
NOMBRE_ALUMNO = 'Luisa Martínez';
```

Los pasos que da Oracle con la sentencia UPDATE (actualizar) son los siguientes:

1. El proceso de usuario pasa la sentencia SQL al proceso del servidor.
2. Los procesos del servidor buscan en el fondo común SQL una versión ejecutable de la sentencia. Si la encuentran, se ejecuta la sentencia SQL; si no la encuentran, se analiza y se guarda su versión ejecutable en el fondo común.
3. Si los datos requeridos no están en la caché de buffers de datos, por medio del proceso DBWR se leen los datos de los archivos de datos y se colocan en los buffers de datos de la SGA.
4. El proceso de servidor coloca los bloqueos en la o las filas afectadas.
5. El proceso de servidor registra los cambios que se van a realizar en el segmento de Roollback y en los datos, y en el buffer de Redo Log creando una transacción.
6. El proceso de servidor registra el valor antiguo de los datos, en bloques de datos de un segmento de Rollback, dentro de la caché de buffers de datos (la nota antigua era 5). El bloque de Rollback se utiliza para almacenar la imagen anterior de los datos, de forma que se puedan deshacer los cambios si fuera necesario. Los segmentos de Rollback existen en los archivos de datos, asociados a *tablespaces* (inicialmente en el *tablespace* de deshacer), mientras que los bloques de Rollback se introducen en el buffer de datos a medida que se necesiten.
7. Se registran los nuevos valores de los datos en los bloques de datos de la caché de buffers para reflejar la nueva nota (7).
8. Cuando hacemos un COMMIT, el LGWR escribe los buffers de Redo Log a los archivos de Redo Log. DBWR escribirá en los archivos de datos todos los bloques que se hayan modificado desde el último punto de control. Se libera la información de deshacer en el segmento de Rollback. Y se liberan los bloqueos de los recursos.



D. Vistas dinámicas de rendimiento

Ya se ha hablado de vistas que contienen información de los archivos de control, estas vistas se denominan **vistas dinámicas de rendimiento** ya que se actualizan constantemente mientras la base de datos permanece abierta y se utilice. Su contenido está relacionado con el rendimiento, y proporcionan datos acerca de las estructuras de disco internas y estructuras de la memoria. Sólo el administrador de base de datos puede acceder a ellas. Los administradores son usuarios con el rol DBA.

Estas vistas se identifican por el prefijo V_$, aunque Oracle proporciona sinónimos públicos con el prefijo V$, la vista V$FIXED_TABLE muestra todas las vistas dinámicas de rendimiento.

Las vistas dinámicas con información de la SGA son las siguientes:

- **V$PARAMETER:** contiene información acerca de los parámetros de inicialización. El comando SHOW PARAMETER muestra los valores actuales de los parámetros de inicialización de la base de datos. Si se desea ver los parámetros que tengan una palabra concreta pondremos SHOW PARAMETER palabra, por ejemplo:

```
SQL> SHOW PARAMETER sga
        NAME                    TYPE          VALUE
    ---------------        -------------     --------
     lock_sga               boolean           FALSE
     pre_page_sga           boolean           FALSE
     sga_max_size           big integer       164M
     sga_target             big integer       0

SQL> SHOW PARAMETER cache_size
        NAME                    TYPE          VALUE
-----------------------   --------------     --------
 db_cache_size              big integer       24M
 db_keep_cache_size         big integer       0
 db_recycle_cache_size      big integer       0
 db_16k_cache_size          big integer       0
 db_2k_cache_size           big integer       0
 db_32k_cache_size          big integer       0
 db_4k_cache_size           big integer       0
 db_8k_cache_size           big integer       0
```

- V$SGA: contiene información resumida sobre la SGA.
- V$OPTION: enumera las opciones que se instalan con el servidor Oracle.
- V$PROCESS: contiene información acerca de los procesos activos actualmente. Si tecleamos `select program from V$process;` veremos los procesos de segundo plano que se ejecutan.
- V$SESSION: enumera la información de la sesión actual.
- V$VERSION: enumera el número de versión y de los componentes.
- V$INSTANCE: muestra el estado de la instancia actual.

E. ¿Qué es una instancia Oracle?

Una **instancia Oracle** es un conjunto de estructuras de memoria (SGA) y procesos en segundo plano que se utilizan para gestionar la base de datos. La instancia sólo puede abrir y utilizar una base de datos a la vez. En la Figura 12.2 se representa el esquema de una instancia Oracle.

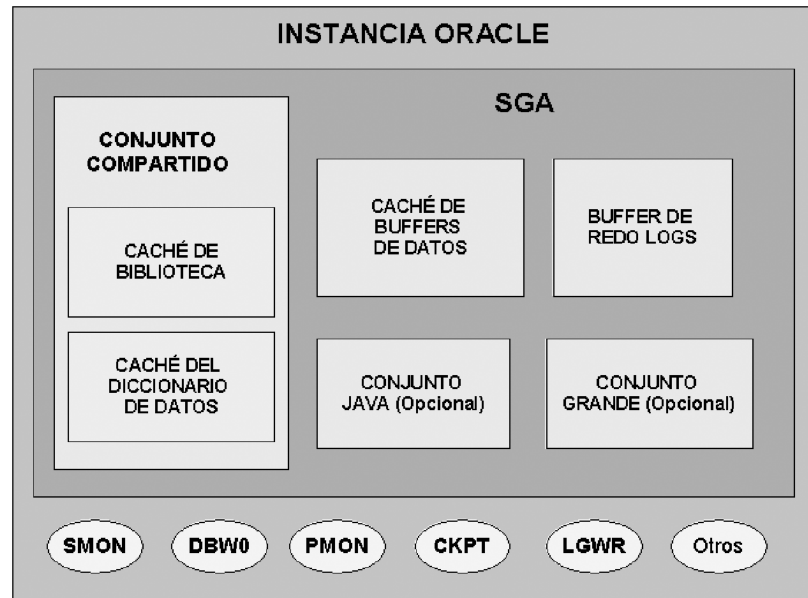


Figura 12.2. *Esquema general de una Instancia Oracle.*

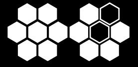
Una base de datos Oracle no está disponible a los usuarios hasta que el administrador de la base de datos haya iniciado la instancia y abierto la base de datos. Cada vez que se abre la base de datos, Oracle realiza los siguientes pasos: el inicio de una instancia, el montaje de la base de datos y la apertura de la misma.

Cada vez que se inicia una instancia, Oracle utiliza un archivo de parámetros que contiene los parámetros de inicialización para asignar espacio de memoria a la SGA e iniciar los procesos en segundo plano. Si se inicia una instancia y se abre la base de datos, los pasos a seguir para cerrar la base de datos son: cerrar la base de datos, desmontarla y cerrar la instancia.

F. Otras estructuras de almacenamiento

Todos los datos de la base de datos están almacenados a nivel lógico en *tablespaces*. Un *tablespace* puede estar contenido, a nivel físico, en uno o más archivos de datos. Existen otras estructuras lógicas de almacenamiento que son las siguientes:

- Los **segmentos** son el espacio que ocupan los objetos de una base de datos. Cuando se crea un objeto, por ejemplo una tabla, automáticamente se crea y asigna un segmento como mínimo. Físicamente se ubican en los archivos de datos.



- Las extensiones son un conjunto de bloques Oracle contiguos. Cuando se crea un *tablespace,* el archivo de datos correspondiente contiene un bloque de cabecera, y una extensión libre formada por el resto del archivo. Pues bien, a medida que se crean segmentos se les asigna espacio de las extensiones libres de un *tablespace*.

- Los **bloques Oracle** son las unidades mínimas de entrada / salida. Están formados por uno o más bloques de datos del Sistema Operativo. El bloque Oracle se define cuando se crea la base de datos en el parámetro: DB_BLOCK_SIZE.

En la Figura 12.3 al margen podemos ver la relación entre las estructuras de almacenamiento, a nivel lógico, de una base de datos.

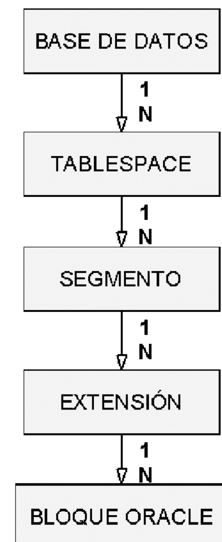


Figura 12.3. *Estructuras de almacenamiento Oracle.*

Los segmentos

Ya hemos visto que al crear un objeto se crea un segmento. Los segmentos están contenidos dentro de los *tablespaces*. Podemos decir que un **segmento** es un conjunto de extensiones de bloques Oracle que pueden estar en varios archivos de un *tablespace*. Cuando se borra un segmento, el espacio ocupado es devuelto al *tablespace*. Cada segmento tiene un conjunto de parámetros de almacenamiento que permiten controlar su crecimiento:

- INITIAL: tamaño de la extensión inicial.

- NEXT: tamaño de la siguiente extensión a asignar.

- MINEXTENTS: número de extensiones asignadas en el momento de la creación del segmento.

- MAXEXTENTS: número máximo de extensiones.

- PCTINCREASE: porcentaje en el que crecerá la siguiente extensión antes de que se asigne, en relación con la última extensión utilizada.

- PCTFREE: porcentaje de espacio libre para actualizaciones de filas que se reserva dentro de cada bloque asignado al segmento.

- PCTUSED: porcentaje de utilización del bloque por debajo del cual Oracle considera que un bloque puede ser utilizado para insertar filas nuevas en él.

- TABLESPACE: nombre del *tablespace* donde se creará el segmento.

Se utilizarán las siguientes vistas para ver estas estructuras de almacenamiento:

- **DBA_SEGMENTS:** información acerca de los segmentos: nombre, propietario, tipo de segmento, *tablespace*, tamaño en extensiones y bloques, las definiciones del almacenamiento, etcétera. La siguiente consulta visualiza los segmentos, número de extensiones y bloques del propietario SCOTT:

```
 SQL> SELECT SEGMENT_NAME,TABLESPACE_NAME,EXTENTS,BLOCKS
FROM DBA_SEGMENTS
    WHERE OWNER='SCOTT';
```

- **DBA_EXTENTS:** información acerca de las extensiones (nombre, propietario, tamaño en bloques y bytes, nombre del *tablespace* etcétera. La siguiente consulta visualiza las extensiones del segmento EMP del propietario SCOTT:

```
SQL> SELECT TABLESPACE_NAME,BLOCKS, BYTES FROM DBA_EXTENTS
WHERE OWNER='SCOTT' AND SEGMENT_NAME='EMP';
```

Tipos de segmentos

- **Segmentos de datos o tablas:** se crean con CREATE TABLE. La tabla es el método más común para almacenar datos.

- **Segmentos de Índices:** CREATE INDEX, todos los registros de un índice determinado se almacenan en un segmento de índices, independiente del de datos. Conviene tener un *tablespace* para los índices distinto al de los datos, para mejorar el rendimiento.

- **Segmentos de Rollback:** son los que permiten efectuar la restauración de las transacciones no validadas, y aseguran la consistencia en lectura. La estructura de un registro de Rollback es la siguiente: Identificador de la transacción, dirección del bloque, número de fila, número de columna, y valor del dato antiguo antes de ser modificado. Para poder realizar operaciones en una base de datos es necesario crear al menos un segmento de Rollback. Cada segmento de Rollback debe tener al menos dos extensiones, reutilizables de manera cíclica. El comportamiento de los segmentos de Rollback es circular: si el número de extensiones ha alcanzado ya al valor del parámetro MAXEXTENTS, una vez que se llega al final de la última extensión, se sobrescribe la primera. Suelen ser objetos compartidos cuyo propietario es PUBLIC. A la hora de realizar una transacción (instrucción de DDL o de DML que no sea una SELECT) se asignan los segmentos de forma automática aunque también se puede asignar un segmento de Rollback particular con la orden: SET TRANSACTION USE ROLLBACK SEGMENT *nombre_segmento_rollback*. Para crear un segmento de Rollback utilizaremos la orden:

```
CREATE [PUBLIC] ROLLBACK SEGMENT nombre_segmento_rb
TABLESPACE nombre_tablespace
STORAGE(
INITIAL tamaño [K|M]
NEXT tamaño [K|M]
OPTIMAL tamaño [K|M]
MINEXTENTS número_ext
MAXEXTENTS número_ext | UNLIMITED);
```

- **Segmentos temporales:** son creados por Oracle para uso cuando debe realizar una ordenación que no le cabe en memoria, en las operaciones: CREATE INDEX, ORDER BY, GROUP BY, DISTINCT, UNION, INTERSECT, MINUS. Se eliminan automáticamente cuando la sentencia finaliza.



12.4 Instalación de Oracle 10g

Antes de trabajar con instancias y bases de datos vamos a proceder a la instalación de Oracle 10g, pero antes de instalar hay que tener en cuenta los requerimientos hardware y software necesarios que soporten la instalación. En la Tabla 12.1 se muestran los requerimientos mínimos, sin embargo, siempre hay que tener un equipo con superiores prestaciones.

REQUERIMIENTO	MÍNIMO VALOR
REQUERIMIENTOS HARDWARE	
Memoria RAM	256 MB, recomendado 512MB
Memoria virtual	Doble de cantidad de la RAM
Espacio en HD	1,5 Gb
Espacio para archivos temporales en HD	100 Mb
Tarjeta de vídeo	256 colores
Procesador	200 MHz
REQUERIMIENTOS SOFTWARE	
Arquitectura del sistema	32-bits, para sistemas operativos de 32 bits. Oracle también dispone de versiones de 64 bits para Windows.
Sistema Operativo	- Windows NT Server 4.0, Windows NT Server Enterprise Edition 4.0, y Terminal Server Edition con service pack 6a o superior. En Windows NT Workstation no se soporta. - Windows 2000 con service pack 1 o superior. - Windows Server 2003. - Windows XP Professional.
Protocolo de red	Oracle Net foundation layer utiliza soporte de protocolo Oracle para comunicarse con los siguientes protocolos de red estándares: - TCP/IP - TCP/IP with SSL - Named Pipes

Tabla 12.1. *Requerimientos mínimos de hardware y software para la instalación de Oracle 10g.*

Si hay otras versiones de Oracle instaladas, conviene detener todas las instancias iniciadas y parar todos los servicios relacionados con Oracle: *Panel de control / Herramientas administrativas / Servicios*.

Para instalar Oracle 10g, introducimos el CD y se inicia la instalación, si no es así se ejecuta el setup.exe. La versión 10g se puede descargar desde la URL:

http://www.oracle.com/technology/software/products/database/Oracle10g/index.html.

La instalación es muy sencilla. Se elige *Instalación Básica*. Especificamos la ubicación dentro del disco duro, y se crea una base de datos inicial durante la instalación, en la que pondremos el nombre de la *Base de Datos Global* y la contraseña para los usuarios SYS, SYSTEM, que son los super-administradores de la base de datos, y SYSMAN y DBSNMP, los super-administradores del *Enterprise Manager*.

Si el equipo está dentro de un dominio de red se pone el nombre de la base de datos y el del dominio.

En la Figura 12.4, se crea la base de datos ORCL. El equipo no forma parte de ningún dominio de red. No hay que olvidar la contraseña para una vez finalizada la instalación conectarnos a la base de datos.

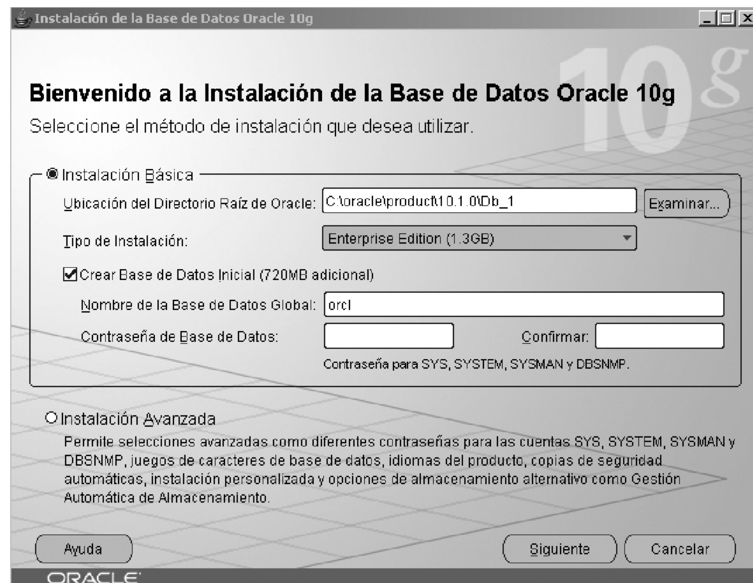


Figura 12.4. *Instalación de Oracle 10g. Paso 1.*

Pulsamos *Siguiente* y nos aparece un resumen de lo que se va a instalar. Se va pulsando *Siguiente* y se van instalando y configurando todos los componentes de Oracle 10g y de la base de datos. Después de los asistentes y configuración de la base de datos se muestra la Figura 12.5 donde se indica la información de la base de datos creada: nombre, identificador del sistema o SID, nombre del archivo de parámetros del servidor. Desde esta ventana se puede acceder a la gestión de contraseñas de los usuarios.

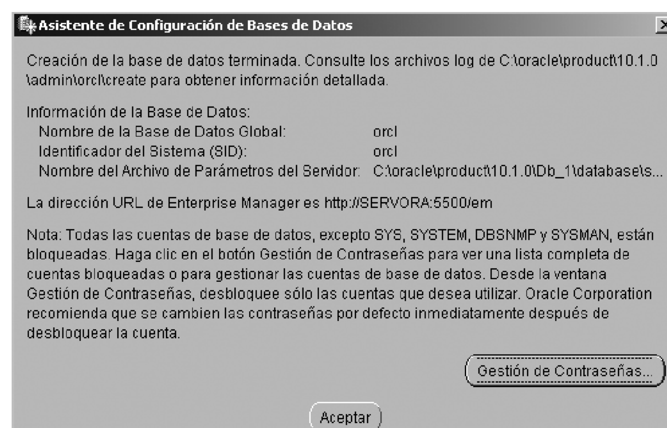
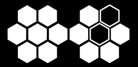


Figura 12.5. *Instalación de Oracle 10g. Información de la base de datos creada.*



Para entrar y gestionar la base de datos y acceder al *Enterprise Manager* abrimos el navegador (si es el Explorer la versión debe ser como mínimo la 6; si es Mozilla, la 1.7 o superior) y tecleamos la dirección indicada después de la instalación, por ejemplo: http://servora:5500/em/ (*Servora* es el nombre de la máquina, también se puede poner http://localhost:5500/em).

Nos conectamos como *sys*. La contraseña es la que indicamos en la instalación, y entramos como SYSDBA. La primera vez pide aceptar la licencia. Una vez aceptada aparece la ventana que se muestra en la Figura 12.6.

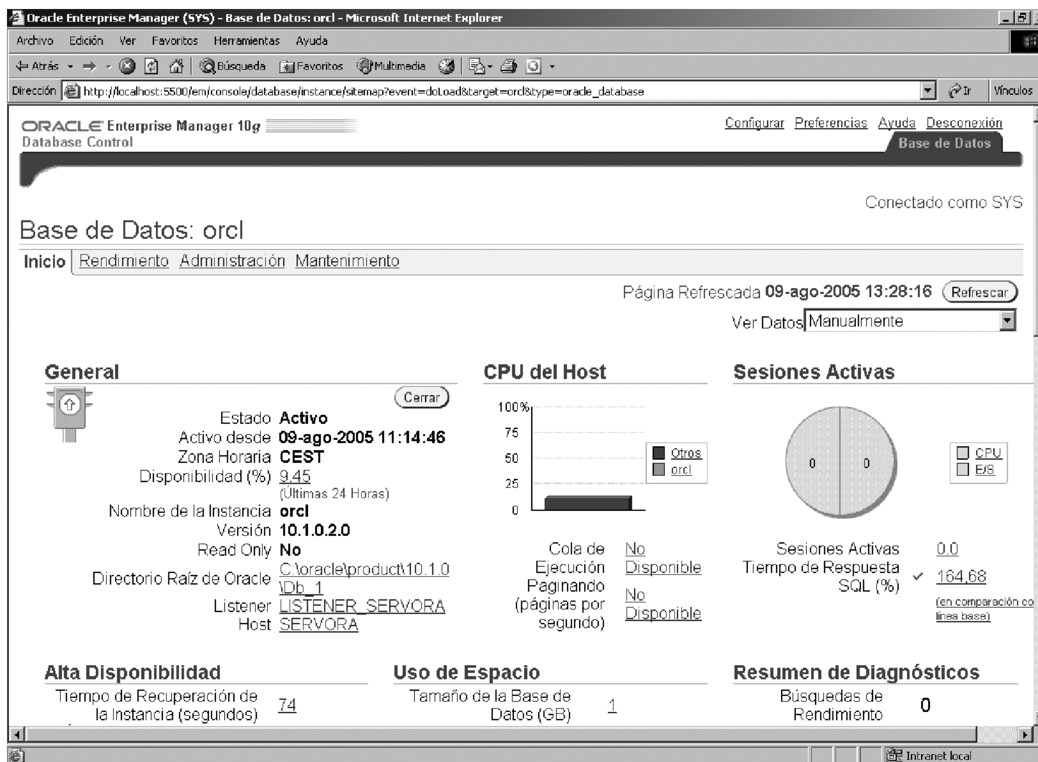


Figura 12.6. *Pantalla inicial del Enterprise Manager.*

Antes de utilizar esta potente herramienta hay que fijarse en las carpetas y la información que se genera en la creación de la base de datos:

– El directorio donde se instala todo el software de Oracle10g, es decir el ORACLE_HOME es: C:\ORACLE\PRODUCT\10.1.0\DB_1

– En el directorio C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\ se encuentran los archivos de datos de la base de datos creada: archivos de control, de espacios de tablas y de redologs.

– En el directorio C:\ORACLE\PRODUCT\10.1.0\ADMIN\ORCL\ se encuentran los archivos de log con la información detallada de las trazas y alertas producidas en la creación y explotación de la base de datos. También se encuentra una carpeta con información del archivo de inicialización.

Cuando arranca la base de datos, Oracle busca el archivo donde se almacenan las características de la instancia Oracle (los **pfile** o **archivos de parámetros**), como el tamaño de las estructuras de memoria en la SGA, el número de procesos en segundo plano, las carpetas de destino de los archivos de traza y alertas, etcétera. Estos parámetros se especifican durante el inicio de la misma.

Antes de la versión 9i, estos parámetros se almacenaban en archivos *init.ora*, que incluian el nombre de la instancia como parte del nombre del archivo. Así, para la base de datos creada el archivo de parámetros es *initorcl.ora*, se puede ver en la carpeta C:\ORACLE\PRODUCT\ADMIN\ORCL\PFILE.

A partir de la versión 9i, se pueden utilizar archivos de parámetros de servidor o spfile. Oracle los crea a nivel de sistema operativo. Este archivo se actualiza automáticamente cuando se utiliza el comando **ALTER SYSTEM.** Para modificar los parámetros de la base de datos. Cuando se inicializa la base de datos Oracle comprueba en primer lugar si existe el archivo *spfilenombre_instancia.ora*, y si no existe busca el archivo *init.ora* tradicional.

El spfile de la base de datos creada, spfileorcl.ora, se encuentra en la carpeta: C:\ORACLE\PRODUCT\10.1.0\DB_1\DATABASE\

Si nos fijamos en el contenido del archivo spfileorcl.ora podemos ver los parámetros de inicialización de una base de datos, en los que hay que destacar:

- DB_NAME: donde se especifica el nombre de la instancia.

- CONTROL_FILES: donde se especifican los nombres de los archivos de control de la base de datos. Si la máquina contiene varios discos, conviene almacenarlos en discos diferentes para prevenir riesgos en soportes físicos. Oracle crea tres archivos, si se desea añadir uno nuevo, se cierra la base de datos, se copia uno de los archivos de control actuales en la nueva ubicación, se modifica el parámetro CONTROL_FILES, y se reinicia la base de datos.

- DB_BLOCK_SIZE: tamaño del bloque de datos Oracle.

- DB_CACHE_SIZE: tamaño de memoria SGA para la caché de buffers de datos.

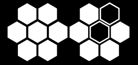
- SHARED_POOL_SIZE: tamaño de memoria SGA para el conjunto compartido donde se compilan las sentencias SQL.

- JAVA_POOL_SIZE: tamaño de memoria SGA para la ejecución de los programas java.

- LARGE_POOL_SIZE: tamaño grande de memoria SGA para utilizar el RMAN, o la opción de servidor compartido o cuando se realizan con frecuencia operaciones de copias de seguridad o restauración.

- SORT_AREA_SIZE: tamaño de memoria que se utiliza para la ordenación de datos solicitados en las sentencias SQL.

- PGA_AGGREGATE_TARGET: tamaño de memoria de la PGA.



- BACKGROUND_DUMP_DEST: ubicación donde se escriben los archivos de rastreo de los procesos en segundo plano: LGWR, DBW, PMON, SMON, etcétera. También se ubica el archivo ALERT (alert_instancia).

- DB_RECOVERY_FILE_DEST: destino para la recuperación de la base de datos.

- USER_DUMP_DEST: destino para los archivos de rastreo de depuración del usuario a favor de un proceso de usuario, cuando los usuarios lo soliciten.

- COMPATIBLE: versión del servidor con el que la instancia es compatible.

- LOG_ARCHIVE_START: si se pone a TRUE se activa la realización de la copia automática.

- REMOTE_LOGIN_PASSWORD_FILE: indica el modo de apertura para el archivo de contraseñas, a partir de la 9i se pone EXCLUSIVE. El archivo de contraseñas se crea y se utiliza para que los usuarios DBA sean autenticados por el sistema operativo.

A. Creación manual de una base de datos Oracle

Oracle proporciona herramientas para crear una base de datos de forma automática, sin embargo, en este apartado vamos a proceder a crear una base de datos de forma manual haciéndola disponible para uso general. Antes de crearla hay que definir el entorno dentro del sistema operativo destino de la instalación.

Oracle Installer, ORADIM y los asistentes de configuración de bases de datos definen las variables en el registro de Windows, así como el registro de la instancia Oracle, como un servicio.

Esto se puede ver editando el registro con la utilidad regedit, desde *Inicio/Ejecutar*, y ver el contenido de la carpeta HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE. Ver figura 12.7.

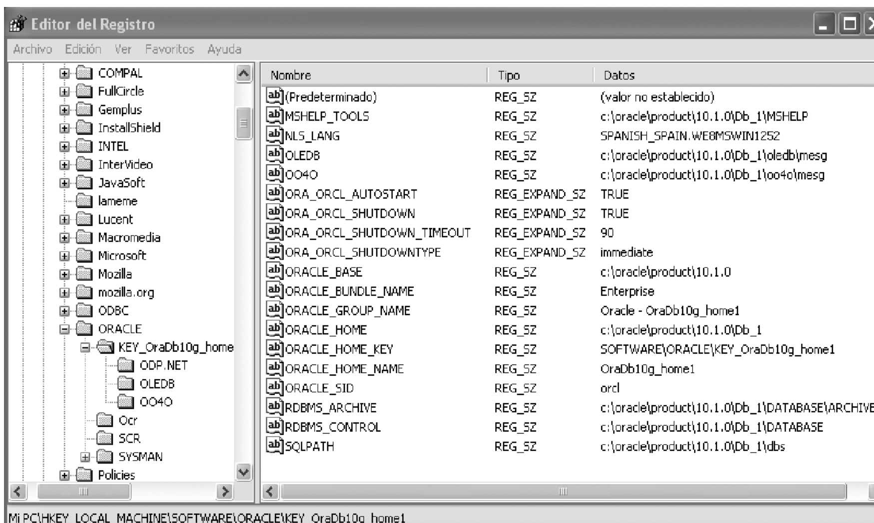


Figura 12.7. *Configuración de Oracle en el registro de Windows.*



Caso práctico

2 **Disponemos de un equipo con capacidad de disco y memoria (1GB como mínimo) para soportar más de una base de datos. El sistema operativo es Windows 2000 y la nueva base de datos se va a llamar ORA2. Ubicamos la base de datos en una nueva carpeta llamada D:\MIBD, que va a contener el archivo de parámetros initora2.ora. Dentro de esta carpeta se crean las siguientes carpetas: D:\MIBD\ORA2\DATA, para los archivos de datos y *tablespaces*, D:\MIBD\ORA2\LOG, para los archivos de Redo Logs, D:\MIBD\ORA2\CONTROL para los archivos de control, y D:\MIBD\ORA2\DUMP para los archivos de traza y alertas.**

Pasos para la creación:

1.º Se crea el archivo de parámetros INITORA2.ORA y se guarda en la carpeta D:\MIBD. El contenido del archivo de parámetros se puede copiar del INITORCL.ORA, para el ejemplo ponemos lo siguiente:

```
*.background_dump_dest='D:\mibd\ora2\dump '
*.compatible='10.1.0.2.0'
*.control_files='D:\mibd\ora2\control\control01.ctl','D:\mibd\ora2\control\control02.ctl','
  D:\mibd\ora2\control\control03.ctl'
*.core_dump_dest='D:\mibd\ora2\dump'
*.db_block_size=8192
*.db_cache_size=25165824
*.db_domain=''
*.db_file_multiblock_read_count=16
*.db_name='ora2'
*.db_recovery_file_dest='D:\mibd\ora2\dump'
*.db_recovery_file_dest_size=2147483648
*.dispatchers='(PROTOCOL=TCP) (SERVICE=ora2XDB)'
*.java_pool_size=50331648
*.job_queue_processes=10
*.large_pool_size=8388608
*.open_cursors=300
*.pga_aggregate_target=25165824
*.processes=150
*.shared_pool_size=83886080
*.sort_area_size=65536
*.user_dump_dest='D:\mibd\ora2\dump'
```

2.º Desde el símbolo del sistema se crea el servicio con ORADIM que va a permitir ejecutar la base de datos. Antes de crear el servicio hay que definir el ORACLE_SID:

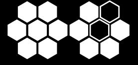
```
C:\>set ORACLE_SID=ORA2
C:\>oradim -new -sid ORA2 -intpwd arm -startmode auto -pfile D:\mibd\initORA2.ora
```

`Intpwd` crea el archivo de contraseñas, con la contraseña *arm* para el usuario SYS, que será el creador de la base de datos. `Startmode` indica el modo de comienzo del servicio, en este caso automático. En `pfile` indicamos la ubicación del archivo de parámetros.

3.º Lo siguiente es entrar desde el símbolo del sistema en sqlplus con el usuario SYS, con la clave asignada en la instalación y conectado como SYSDBA.

```
C:\>sqlplus sys/arm as SYSDBA
```

(Continúa)



(Continuación)

4.º A continuación se inicia la instancia con el comando STARTUP. Este comando admite las siguientes opciones:

– OPEN: abre la base de datos y permite a los usuarios conectarse.

– MOUNT: monta la base de datos para ciertas actividades del DBA, y no permite que los usuarios accedan.

– NOMOUNT: crea la SGA e inicia los procesos en segundo plano, y no permite que los usuarios accedan.

– PFILE: permite el uso de un archivo de parámetros no por defecto, para configurar la instancia.

En el ejercicio iniciamos la instancia en modo NOMOUNT, así pues tecleamos lo siguiente STARTUP NOMOUNT PFILE=D:\MIBD\initORA2.ora. Si esto da error es porque la base de datos está ejecutándose. La cerramos tecleando *shutdown* y volvemos a montar la instancia:

```
SQL> STARTUP NOMOUNT PFILE=D:\MIBD\initORA2.ora
ORA-01081: cannot start already-running ORACLE - shut it down first
SQL> shutdown
ORA-01507: database not mounted
ORACLE instance shut down.
SQL> STARTUP NOMOUNT PFILE=D:\MIBD\initORA2.ora
ORACLE instance started.

Total System Global Area 171966464 bytes
Fixed Size          787988 bytes
Variable Size       145750508 bytes
Database Buffers     25165824 bytes
Redo Buffers        262144 bytes
```

5.º Una vez iniciada la instancia, el siguiente paso es crear la base de datos con el comando CREATE DATABASE. Las opciones más comunes para este comando son las siguientes:

```
CREATE DATABASE nombre_de_base_de_datos

[MAXLOGFILES entero]
  [MAXLOGMEMBERS entero]
  [MAXDATAFILES entero]
  [MAXLOGHISTORY entero]
  [MAXINSTANCES entero]
  [ARCHIVELOG | NOARCHIVELOG]
  [LOGFILE [GROUP entero] especificación del archivo
      [, [GROUP entero] especificación del archivo] . . .]
  [DATAFILE especificación de archivo [autoextend_clause]
     [, [especificación de archivo [autoextend_clause] . . .] ]
  [SYSAUX DATAFILE especificación de archivo [autoextend_clause] ]
        [UNDO TABLESPACE nombre DATAFILE especificación de archivo
[autoextend_clause] ]
        [DEFAULT TEMPORARY TABLESPACE nombre TEMPFILE especificación
de archivo [Temp_autoextend_clause] ]
[CHARACTER SET charset]
[NATIONAL CHARACTER SET charset]
```

(Continúa)

(Continuación)

Donde:

- MAXLOGFILES: es el número máximo de grupos de archivos de Redo Log que se pueden crear para la base de datos.

- MAXLOGMEMBERS: es el número máximo de miembros de archivos de Redo Log que puede tener un grupo.

- MAXDATAFILES: el tamaño de la sección de archivos de datos del archivo de control durante la ejecución de una instrucción CREATE DATABASE o CREATE CONTROLFILE.

- ARCHIVELOG | NO ARCHIVELOG, la segunda es la opción por defecto. Establece si los archivos de Redo Logs online se guardan antes de volverlos a utilizar, modo ARCHIVELOG, o se reutilizan sin guardarlos, modo NOARCHIVELOG. Esta segunda opción no va a permitir que se lleve a cabo una recuperación desde soporte físico.

- MAXLOGHISTORY: este parámetro resulta útil si la base de datos está en modo ARCHIVELOG. Este valor determina cuánto espacio del archivo de control hay que asignar para los nombres de los archivos de Redo Logs archivados.

- MAXINSTANCES: número máximo de instancias que puede montar y abrir simultáneamente la base de datos.

- LOGFILE GROUP: especifica los nombres de los archivos de Redo Log que se deben utilizar y el grupo al que pertenecen.

- DATAFILE: especifica los nombres de los archivos de datos que se va a utilizar, estos archivos formarán parte del *tablespace* SYSTEM.

- SYSAUX DATAFILE: especifica los nombres de los archivos de datos de SYSAUX que se van a utilizar. Estos archivos formarán parte del *tablespace* SYSAUX.

- CHARACTER SET: especifica el juego de caracteres que utiliza la base de datos para almacenar datos.

- NATIONAL CHARACTER SET: especifica el juego de caracteres nacional que utiliza para almacenar datos en columnas definidas como NCHAR, NCLOB o NVARCHAR2. Si no se pone el juego de caracteres será el mismo que el CHARACTER SET.

- Especificación de archivo: indicamos el nombre de archivo, el tamaño en Kilobytes o Megas y la opción REUSE, en este caso el archivo debe existir. Por ejemplo:

```
'D:\MIBD\ora2\DATA\system01.ORA' SIZE 200 M REUSE
```

- Autoextend Clause: activa o desactiva la ampliación automática de un archivo de datos o temporal nuevo. En el ejemplo se amplía automáticamente de 10 en 10 Megas:

```
'D:\MIBD\ora2\DATA\system01.ORA' size 200 M REUSE AUTOEXTEND ON NEXT 10 M
```

La base de datos a crear en el ejercicio es la siguiente:

```
SQL> create database ORA2
   maxlogfiles 32
   maxlogmembers 5
   maxdatafiles 30
   maxloghistory 100
    logfile
        group 1 ('D:\MIBD\ora2\LOG\redo0101.log') size 10M,
```

(Continúa)



(Continuación)

```
          group 2 ('D:\MIBD\ora2\LOG\redo0201.log') size 10 M
datafile 'D:\MIBD\ora2\DATA\system01.ORA' size 200 M AUTOEXTEND ON
SYSAUX datafile 'D:\MIBD\ora2\DATA\sysaux01.ORA' size 200 M
character set we8iso8859p1;
```

La creación de la base de datos puede fallar por las siguientes causas:

- Errores en la sintaxis al escribir las órdenes.
- Los archivos que se van a crear ya existen.
- Errores del sistema como permisos en directorios o archivos, o espacio insuficiente en disco.

Ya se ha visto que para cerrar la instancia se utiliza la orden shutdown, el formato es:

```
SHUTDOWN [NORMAL | TRANSACTIONAL | IMMEDIATE | ABORT ]
```

- La opción por defecto es NORMAL, en el que no se admiten conexiones nuevas, el servidor Oracle espera a que se desconecten todos los usuarios antes de completar el cierre, cierra y desmonta la base de datos y por último cierra la instancia.
- El cierre TRANSACCIONAL evita que los clientes pierdan trabajo: ningún cliente puede iniciar una transacción nueva y el cliente se desconecta cuando termina su transacción en curso.
- En el cierre IMMEDIATE, no se completan las sentencias SQL actuales, el servidor no espera a que los usuarios se desconecten, Oracle deshace las transacciones activas y desconecta a todos los usuarios.
- La opción ABORT se utiliza si el cierre inmediato y normal no funciona.

En entornos Windows se puede parar la base de datos deteniendo los servicios asociados a la instancia.

6.º El siguiente paso es crear todas las tablas y vistas que van a formar el diccionario de datos. El diccionario de datos proporciona información acerca de:

- Las definiciones de todos los objetos de esquema en la base de datos (tablas, vistas, índices, aguzamientos, secuencias, procedimientos, funciones, paquetes, disparadores, etcétera).
- Definiciones y asignaciones de espacio de los objetos de esquema.
- Valores por defecto de las columnas.
- Información acerca de las restricciones de integridad.
- Los nombres de los usuarios Oracle, los privilegios y roles otorgados a cada usuario.
- Auditoría de información como quién ha accedido o actualizado algún objeto del esquema.

El diccionario se crea a partir de los siguientes archivos de comandos ubicados en C:\ORACLE\PRODUCT\10.1.0\DB_1\RDBMS\ADMIN\:

- Catalog.sql: crea las vistas de diccionario de datos y sinónimos de los datos que se utilizan frecuentemente.

(Continúa)

(Continuación)

– Catproc.sql: ejecuta los archivos de comandos necesarios para el PL/SQL del servidor.

Para ejecutar estos archivos de comandos la base de datos tiene que estar en modo OPEN, así pues, desde SQL, la cerramos, la volvemos a abrir y se lanzan los scripts:

```
SQL> shutdown
Database closed.
Database dismounted.
ORACLE instance shut down.
SQL> startup open PFILE=D:\MIBD\initORA2.ora
ORACLE instance started.
Total System Global Area 171966464 bytes
Fixed Size          787988 bytes
Variable Size       145750508 bytes
Database Buffers      25165824 bytes
Redo Buffers        262144 bytes
Database mounted.
Database opened.

SQL> start C:\Oracle\product\10.1.0\db_1\rdbms\admin\catalog.sql;
SQL> start C:\Oracle\product\10.1.0\db_1\rdbms\admin\catproc.sql;
```

Existen varias categorías de archivos de comandos administrativos que normalmente se ejecutan como usuario SYS, destacan:

– Cat*.sql: que crean vistas del diccionario de datos.

– Dbms*.sql: que crean objetos para los paquetes Oracle predefinidos que amplían la funcionalidad del servidor Oracle.

– Utl*.sql: crean vistas y tablas para las utilidades de la base de datos, por ejemplo, el archivo de comando utlxplan.sql crea una tabla que se utiliza para ver el plan de ejecución de una sentencia SQL.

7.º Al terminar los procedimientos podemos listar los usuarios que se han creado.

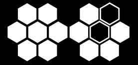
```
SQL> select * from all_users;

      USERNAME           USER_ID        CREATED
 -------------------     ----------     --------
      DBSNMP                  22        02/08/05
      DIP                     19        02/08/05
      OUTLN                   11        02/08/05
      SYSTEM                   5        02/08/05
      SYS                      0        02/08/05
   5 rows selected.
```

A continuación, creamos un usuario y le damos permiso para conectarse a la base de datos creada. Recuerda que antes de realizar transacciones con este usuario sería necesario crear un *tablespace* y segmento de Rollback.

```
SQL> CREATE USER ALICIA IDENTIFIED BY ALICIA;
SQL> GRANT CONNECT TO ALICIA;
```

(Continúa)



(Continuación)

Para probar la conexión de este usuario a la instancia ORA2, hay que crear la cadena de conexión dentro del archivo *tnsnames.ora* que se encuentra en DB_1\NETWORK\ADMIN\. Antes de modificar conviene hacer una copia de este archivo, lo editamos y añadimos:

```
ORA2 =
 (DESCRIPTION =
  (ADDRESS_LIST =
   (ADDRESS = (PROTOCOL = TCP)(HOST = servora)(PORT = 1521))
  )
  (CONNECT_DATA =
   (SERVICE_NAME = ora2)
  )
 )
```

Nos conectamos con SQL Plus, desde Windows y en *Host string* ponemos *ora2* (da lo mismo en mayúscula que minúscula), ver Figura 12.8. Normalmente se visualiza algún mensaje de error, indicando que falta algún paquete, sería necesario ejecutar más archivos de comandos.

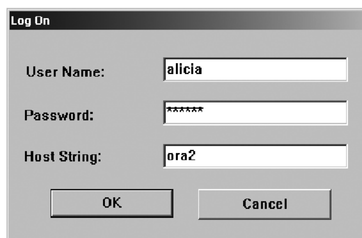


Figura 12.8. *Conexión a la base de datos creada.*

Para que la instancia arranque al iniciar el equipo debemos poner el inicio del servicio en modo automático: *Inicio/Panel de control/Herramientas administrativas/Servicios*. Ver Figura 12.9.

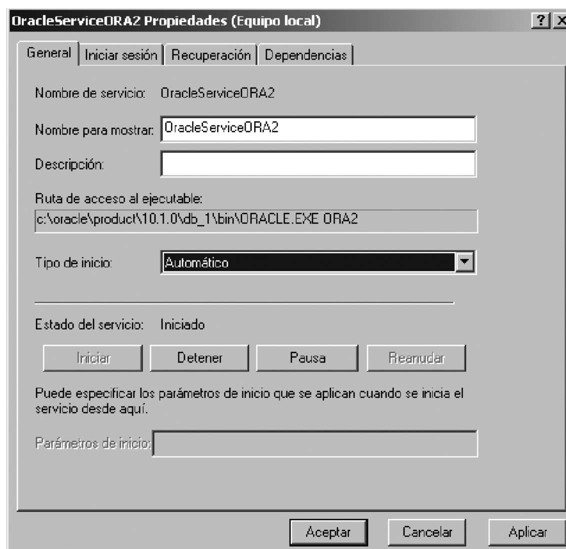


Figura 12.9. *Inicio del servicio en modo automático.*

B. Etapas en el inicio y cierre de la base de datos

Cuando se inicia la base de datos se debe elegir el estado en el que se inicia. Las etapas que se desarrollan son las siguientes:

- **Inicio de la instancia.** Una instancia se inicia sin montar una base de datos sólo durante la creación de la base de datos o la nueva creación de archivos de control. El inicio de la instancia incluye: lectura del archivo de parámetros initsid.ora, asignación de la SGA, inicio de los procesos en segundo plano y apertura del archivo ALERT y los archivos de rastreo.

- **Montaje de la base de datos.** Para realizar las operaciones de mantenimiento específicas, se inicia una instancia y se monta una base de datos, pero no se abre. La base de datos se debe montar pero no abrir durante: cambios de los archivos de datos, la activación y desactivación de las opciones de archivado Redo Log y la recuperación completa de la base de datos.

  El montaje de la base de datos incluye las siguientes tareas: asociación de una base de datos con una instancia iniciada previamente, ubicación y apertura de los archivos de control especificados en el archivo de parámetros, y lectura de los archivos de control para conseguir el nombre y el estado de los archivos de datos y archivos de Redo Log.

- **Apertura de la base de datos.** La apertura permite que cualquier usuario se conecte a la base de datos y realice operaciones normales de acceso a datos. Incluye la apertura de los archivos de datos y la apertura de los archivos de Redo Log. Durante esta etapa, el servidor Oracle comprueba que todos los archivos de datos y de Redo Log se puedan abrir, y comprueba la consistencia de la base de datos y si fuera necesario el SMON inicia la recuperación del sistema.

En la Figura 12.10 se muestran las etapas del inicio de una base de datos.

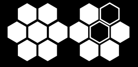
STARTUP PFILE	STARTUP MOUNT	STARTUP NOMOUNT	NOMOUNT	• Lee el Init.ora • Genera la SGA • Inicia la instancia
		ALTER DATABASE MOUNT	MOUNT	• Abre archivos de control para esa instancia.
		ALTER DATABASE OPEN	OPEN	• Abre los archivos de datos y redo log. • Da servicio a todos los usuarios

Figura 12.10. *Etapas de inicio de una base de datos.*

A la hora de cerrar una instancia se realizan los siguientes pasos:

- **Cierre de la base de datos.** El servidor Oracle escribe los cambios de las caché de buffers de datos y los registros de la caché de buffers de Redo Logs en los archivos de datos y los archivos de Redo Logs. Seguidamente Oracle cierra los archivos de datos y de Redo Logs. Los archivos de control permanecen abiertos.

- **Desmontaje de la base de datos.** Al desmontar la base de datos el servidor Oracle cierra los archivos de control.



- **Cierre de la instancia.** Es el último paso, cuando se cierra la instancia se cierra el archivo ALERT y los archivos de rastreo, se libera la SGA y se terminan los procesos en segundo plano.

Comando STARTUP

```
STARTUP [FORCE] [RESTRICT] [PFILE=nombrearchivo]
[OPEN [RECOVER] [basedatos] | MOUNT | NOMOUNT ]
```

`FORCE`: aborta la instancia en ejecución antes de realizar un inicio normal.

`RESTRICT`: sólo permite que los usuarios con el privilegio RESCTRICTED SESSION accedan a la base de datos.

`RECOVER`: comienza la recuperación del medio físico cuando se inicia la base de datos.

Comando ALTER DATABASE

```
ALTER DATABASE { MOUNT | OPEN [READ WRITE | READ ONLY] }
```

`READ WRITE`: abre la base de datos en modo lectura escritura. Es la opción por defecto. Permite generar los Redo Logs.

`READ ONLY`: sólo se permiten operaciones de lectura evitando que los usuarios generen Redo Log.

E. Vistas del diccionario de datos

En el Caso práctico 2 hemos visto la creación de una base de datos y las vistas del diccionario de datos. Estas vistas resumen y muestran la información almacenada en las tablas base, las que almacenan información de la base de datos. Existen varias categorías de vistas del diccionario:

- **Vistas con el prefijo DBA:** muestran una vista global de toda la base de datos, su finalidad es que la consulten sólo los administradores. Cualquier usuario con el privilegio del sistema SELECT ANY TABLE podrá consultar las vistas con prefijo DBA.

- **Vistas con el prefijo ALL:** estas vistas hacen referencia a la perspectiva global del usuario de la base de datos. Devuelven información acerca de los objetos de esquema a los que el usuario tiene acceso.

- **Vistas con el prefijo USER:** estas vistas hacen referencia a lo que hay en el esquema del usuario, es decir, a los objetos que posee el usuario.

Para obtener una visión general del diccionario de datos se puede consultar la vista DICTIONARY o su sinónimo DICT, teclea SELECT TABLE_NAME FROM DICT ORDER BY TABLE_NAME; para ver los nombres de las tablas del diccionario de datos de la base de datos.

F. Creación de una base de datos utilizando el asistente

Oracle proporciona herramientas para crear, cambiar la configuración y borrar una base de datos de forma asistida, también se podrá crear una base de datos de una lista de plantillas predefinidas. Para ejecutar el asistente elegimos: *Inicio/programas/Oracle-OraDb10g_Home1/Configuration and Migration Tools/Database Configuration Assistant*. Aparecerá una pantalla de bienvenida. Pulsamos *Siguiente* y seguidamente pedirá elegir entre las siguientes opciones:

- ***Crear una base de datos:*** permite crear una nueva base de datos o una plantilla.
- ***Configurar opciones de base de datos:*** permitirá cambiar la configuración y agregar opciones que no se hayan configurado previamente.
- ***Suprimir una base de datos:*** se borrarán todos los archivos de datos.
- ***Gestionar plantillas:*** las plantillas se utilizan para guardar la definición de una base de datos de forma que se puedan crear bases de datos duplicadas sin tener que especificar los parámetros de nuevo.

Si se elige *Crear una base de datos*, aparecerá una ventana (ver Figura 12.11) en la que se elegirá de *Uso General*. La opción *Almacén de datos* se utiliza para crear bases de datos encaminadas al *Data Mining*. La opción *Personalizar base de datos* permite crear una base de datos personalizada. *Procesamiento de transacciones* se utiliza para crear bases de datos cuyo cometido es el proceso de transacciones. Dependiendo del tipo los parámetros de creación cambian.

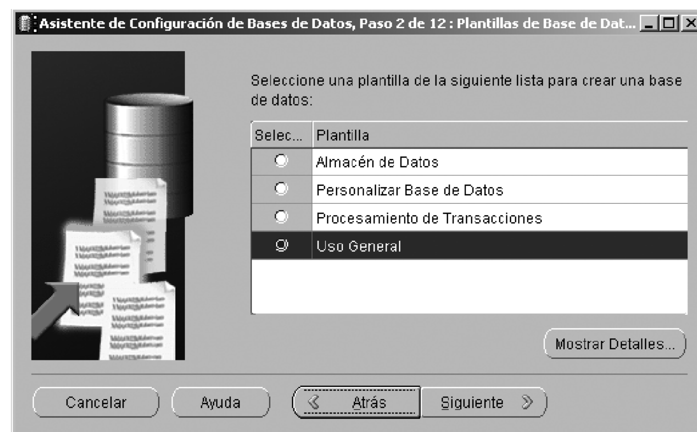
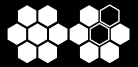


Figura 12.11. *Elección de la base de datos a crear.*

En las siguientes pantallas que aparecen teclearemos los datos que se piden: nombre y SID de la base de datos. Las opciones de gestión que se desea utilizar, dejaremos las opciones por defecto: *Configurar Base de Datos con Enterprise Manager*, y usar *Database Control* para gestión de la base de datos. Las contraseñas para los usuarios: SYS, SYSTEM, DBSMNP y SYSMAN. En *Mecanismo de almacenamiento* se deja el *Sistema de archivos*. En las ubicaciones para los archivos de datos, las indicadas en la plantilla por defecto. Se indican las opciones de recuperación de archivos, el directorio y el tamaño del área de



recuperación. Se podrán ver las variables de ubicación de archivos pulsando al botón correspondiente (ver Figura 12.12).

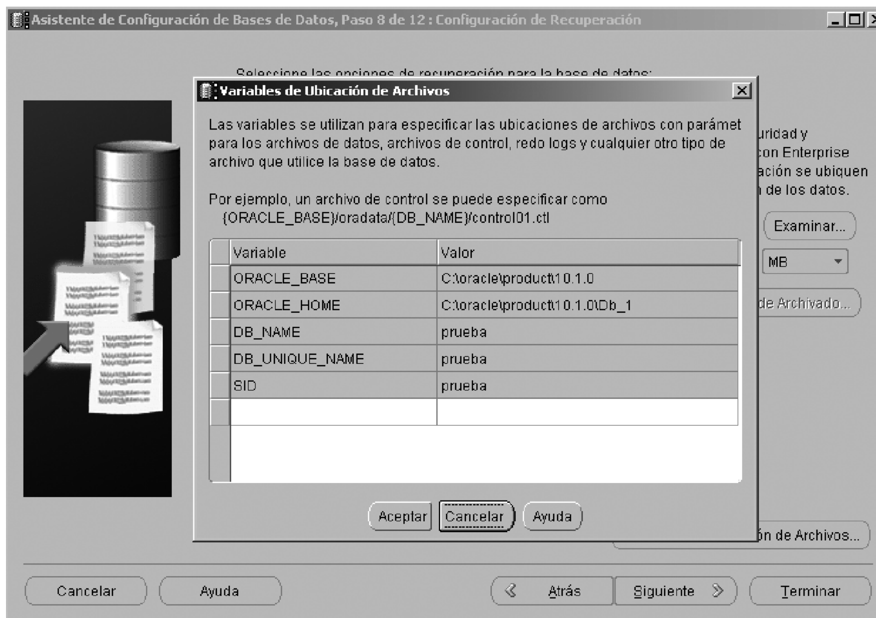


Figura 12.12. *Variables de ubicación de archivos.*

En la creación también pedirá si se desea incluir los esquemas ejemplo o si se desea ejecutar algún archivo de comandos.

En la siguiente pantalla se muestran tres pestañas en las que hay que indicar los parámetros de inicialización de la memoria SGA, el tamaño del bloque de datos, el número de procesos de usuario de sistema operativo que se pueden conectar a la vez, el juego de caracteres y el modo de conexión (ver Figura 12.13).

Finalmente pedirá que se cree la base de datos.

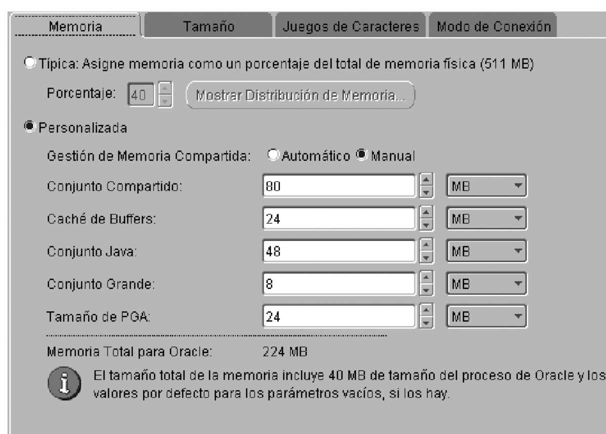


Figura 12.13. *Parámetros de la base de datos.*



Caso práctico

3 **En este caso práctico vamos a crear una base de datos personalizada, la llamaremos PRUEBA. Generando los scripts necesarios para su creación, los lanzaremos para crear la base de datos. Así pues, se inicia el asistente como se indicaba anteriormente pero se elige *Personalizar base de datos*.**

Los pasos son casi idénticos, hay que prestar atención al paso 12 del asistente en el que se indica el directorio destino para ubicar los scripts. En esta ventana dejaremos en blanco la casilla de creación e indicaremos una carpeta para guardar los scripts (ver Figura 12.14).

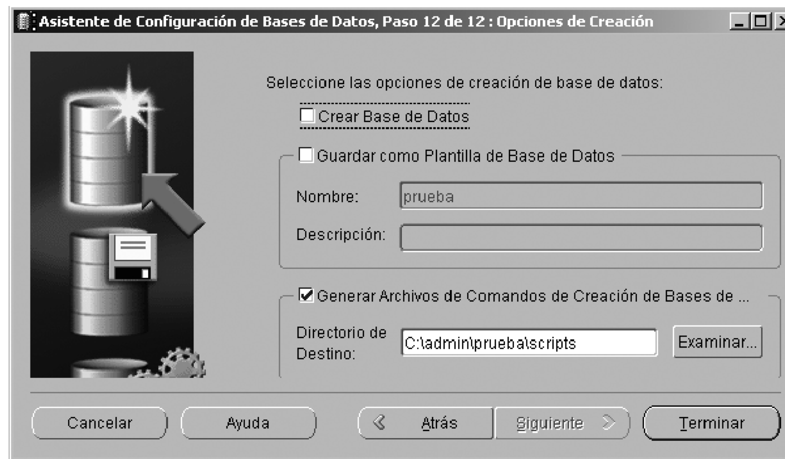
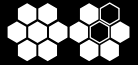


Figura 12.14. *Ubicación de scripts para la creación de la base de datos.*

Una vez creados abrimos la carpeta y examinamos los archivos creados. Observa el archivo por lotes prueba.bat. Es el primero que se debe de lanzar desde la interface de comandos y éste crea los directorios para la ubicación de los archivos de datos, el servicio para iniciar la instancia con oradim. Luego lanza el script `prueba.sql` desde el que se irán ejecutando el resto de scripts. A continuación, se muestra el contenido de prueba.bat:

```
mkdir C:\Oracle\product\10.1.0\admin\prueba\bdump
mkdir C:\Oracle\product\10.1.0\admin\prueba\cdump
mkdir C:\Oracle\product\10.1.0\admin\prueba\create
mkdir C:\Oracle\product\10.1.0\admin\prueba\pfile
mkdir C:\Oracle\product\10.1.0\admin\prueba\udump
mkdir C:\Oracle\product\10.1.0\Db_1\database
mkdir C:\Oracle\product\10.1.0\flash_recovery_area
mkdir C:\Oracle\product\10.1.0\oradata\prueba
set ORACLE_SID=prueba
C:\Oracle\product\10.1.0\Db_1\bin\oradim.exe -new -sid PRUEBA -startmode
manual -spfile
C:\Oracle\product\10.1.0\Db_1\bin\oradim.exe -edit -sid PRUEBA -
startmode auto -srvcstart system
C:\Oracle\product\10.1.0\Db_1\bin\sqlplus /nolog
@C:\admin\prueba\scripts\prueba.sql
```



12.5 Gestión de seguridad

La gestión de seguridad tiene mucho que ver con la gestión de usuarios y con la concesión y supresión de privilegios a los usuarios. El administrador de la base de datos es el responsable de permitir o denegar el acceso a los usuarios a determinados objetos o recursos de la base de datos.

Podemos clasificar la seguridad de la base de datos en dos categorías: *seguridad del sistema* y *seguridad de los datos*.

- La **seguridad del sistema** incluye los mecanismos que controlan el acceso y uso de la base de datos a nivel del sistema. Por ejemplo: cada vez que se conecta un usuario a la base de datos, los mecanismos de seguridad comprobarán si éste está autorizado.

- La **seguridad de los datos** incluye mecanismos que controlan el acceso y uso de la base de datos a nivel de objetos. Por ejemplo, cada vez que un usuario acceda a un objeto (una tabla, una vista, etcétera), los mecanismos de seguridad comprobarán si dicho usuario puede acceder a ese objeto y qué tipo de operación puede hacer con él (SELECT, INSERT, etcétera).

A. Usuarios

Un **usuario** es un nombre definido en la base de datos que se puede conectar a ella y acceder a determinados objetos según ciertas condiciones que define el administrador. Para acceder a la base de datos, los usuarios deben ejecutar una aplicación de base de datos, como SQL*Plus, Oracle Forms y conectarse usando el nombre definido en la base de datos. También pueden acceder a través de un navegador web.

Estos mecanismos de seguridad no son más que privilegios del sistema, privilegios sobre objetos, roles, límites a los recursos, asignación de espacio para almacenar objetos, etc.

Asociado con cada usuario de la base de datos hay un esquema con el mismo nombre. Un **esquema** es una colección lógica de objetos (tablas, vistas, secuencias, sinónimos, índices, clusters, procedures, funciones, paquetes, etcétera). Por defecto, cada usuario tiene acceso a todos los objetos de su esquema correspondiente, y puede acceder a los objetos de otro usuario siempre y cuando este otro le haya concedido el privilegio de hacerlo. Veamos cómo se crean, se modifican y borran los usuarios.

- **Creación de usuarios**

Al instalar la base de datos Oracle se crean automáticamente dos usuarios con el privilegio de administrador de la base de datos (DBA). Estos usuarios son SYS y SYSTEM. Durante el proceso de instalación se pedirá la clave para cada usuario.

El usuario SYS es el propietario de las tablas del diccionario de datos; en ellas se almacena información sobre el resto de las estructuras de la base de datos. Oracle maneja las tablas del usuario SYS; ningún usuario, aunque sea administrador, puede modificar las tablas de SYS. Sólo nos conectaremos como usuario SYS cuando las instrucciones de Oracle lo exijan.

El diccionario de datos está formado por un conjunto de tablas y vistas en el *tablespace* SYSTEM. Los usuarios tienen acceso de sólo lectura a las vistas de este diccionario, el cual se crea a la vez que la base de datos y es propiedad del usuario SYS. Contiene: objetos de la base

de datos, nombres de usuario, derechos y autorizaciones, restricciones, información sobre el espacio libre/ocupado, información de exportación e información sobre otros objetos.

Los objetos del diccionario de datos a los que un usuario puede acceder se encuentran en la vista DICTIONARY, que es propiedad del usuario SYS. Con la orden `SQL> SELECT TABLE_NAME FROM DICTIONARY;` se visualizan los objetos del diccionario de datos a los que se puede acceder. El prefijo de una vista del diccionario indica el nivel de acceso a él:

Vistas USER y ALL	Accesibles para todos los usuarios.
Vistas DBA	Sólo el administrador puede utilizar estas vistas.

El usuario SYSTEM es creado por Oracle para realizar las tareas de administración de la base de datos. No se suelen crear tablas de usuario en el esquema de SYSTEM. Para crear otros usuarios es preciso conectarse como usuario SYSTEM, ya que éste posee el correspondiente privilegio. Al instalar Oracle, el administrador de la base de datos ha de crearse un usuario para sí mismo con los derechos de administrador y realizar todas las tareas de administración con este nombre de usuario.

Para crear usuarios se necesita el privilegio CREATE USER. La orden para crear usuarios es CREATE USER, cuyo formato es:

```
CREATE USER nombre_usuario
IDENTIFIED BY clave_acceso
[DEFAULT TABLESPACE espacio_tabla]
[TEMPORARY TABLESPACE espacio_tabla]
[QUOTA {entero {K|M} |UNLIMITED} ON espacio_tabla]
[PROFILE perfil];
```

Donde:

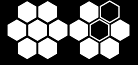
- La primera opción, CREATE USER, crea un nombre de usuario que será identificado por el sistema.

- La segunda opción, IDENTIFIED BY, permite dar una clave de acceso al usuario creado. Por ejemplo, para crear el usuario MILAGROS, con clave de acceso MILAGROS, utilizaremos la siguiente orden: `CREATE USER MILAGROS IDENTIFIED BY MILAGROS;` Pero si MILAGROS intenta conectarse a Oracle no podrá porque no tiene privilegios para iniciar sesión en la base de datos. Oracle devuelve un mensaje de error al intentar conectar:

```
SQL> CONNECT
Introduzca el nombre de usuario: MILAGROS
Introduzca la contraseña: ********
ERROR:
ORA-01045: user MILAGROS lacks CREATE SESSION privilege;
logon denied
```

- DEFAULT TABLESPACE asigna a un usuario el *tablespace* por defecto para almacenar los objetos que cree. Si no se asigna ninguno, el *tablespace* por defecto será USERS.

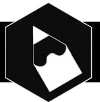
- TEMPORARY TABLESPACE especifica el nombre de *tablespace* para trabajos temporales. Si no se especifica ninguno, el *tablespace* por defecto es TEMP.



- QUOTA asigna un espacio en megabytes o en kilobytes en el *tablespace* asignado. Si no se especifica esta cláusula, el usuario no tiene cuota asignada y no podrá crear objetos en el *tablespace*. Si un usuario dispone de acceso y de recursos ilimitados a cualquier *tablespace*, se le debe dar el privilegio UNLIMITED TABLESPACE con la orden GRANT, que se verá después.

- PROFILE asigna un perfil al usuario. Si se omite, Oracle asigna el perfil por omisión al usuario. Un perfil limita el número de sesiones concurrentes de usuario, limita el tiempo de uso de la CPU, el tiempo de una sesión, desconecta al usuario si sobrepasa el tiempo, etcétera.

Caso práctico



4 **La siguiente orden crea un usuario de nombre USUARIO1. La clave es la misma. El *tablespace* por omisión es USERS (ya que no se indica) al cual se han asignado 500 kilobytes. El *tablespace* para trabajos temporales es TEMP (ya que no se indica en la orden):**

```
CREATE USER USUARIO1 IDENTIFIED BY USUARIO1 QUOTA 500K ON USERS;
```

La siguiente orden crea un usuario de nombre USUARIO2, la clave es la misma. El *tablespace* por omisión es TRABAJO al cual se han asignado 1 megabyte. El *tablespace* para trabajos temporales es TEMPORAL al cual se han asignado 500 kilobytes:

```
CREATE USER USUARIO2 IDENTIFIED BY USUARIO2 DEFAULT TABLESPACE TRABAJO TEMPORARY
TABLESPACE TEMPORAL QUOTA 1M ON TRABAJOS QUOTA 500K ON TEMPORAL;
```

Vistas con información de usuarios:

- **USER_USERS:** Obtiene información del usuario actual: la fecha de creación, los *tablespaces* asignados, el identificador, etcétera.

- **ALL_USERS:** Obtiene información acerca de todos los usuarios creados en la base de datos: el nombre, la fecha de creación y su identificador.

Modificación de usuarios

Las opciones dadas a un usuario en la orden CREATE USER se pueden modificar con a orden ALTER USER. Es posible cambiar la clave de acceso, el *tablespace* por defecto, el *tablespace* temporal, la cuota en los *tablespaces* o el perfil. El formato es el siguiente:

```
ALTER USER nombre_usuario
IDENTIFIED BY clave_acceso
[DEFAULT TABLESPACE espacio_tabla]
[TEMPORARY TABLESPACE espacio_tabla]
[QUOTA {entero {K|M} |UNLIMITED} ON espacio_tabla]
[PROFILE perfil];
```

Los parámetros significan lo mismo que en la orden CREATE.

Cada usuario puede cambiar únicamente su clave de acceso. No puede cambiar el *tablespace* por defecto, la cuota en los *tablespaces* ni el perfil, a no ser que tenga el privilegio ALTER USER. Por ejemplo, el USUARIO1 cambia su clave: **ALTER USER** USUARIO1 IDENTIFIED BY NUEVACLAVE;

Borrado de usuarios

Podemos borrar un usuario de la base de datos, incluidos los objetos que contiene. Para borrar usuarios se usa la orden DROP USER, que tiene este formato:

```
DROP USER usuario [CASCADE];
```

La opción CASCADE suprime todos los objetos del usuario antes de borrar el usuario.

Para poder borrar usuarios es preciso tener el privilegio DROP USER.

Por ejemplo, borramos el USUARIO1: **DROP USER** USUARIO1 . Si el usuario tiene objetos creados se visualiza un mensaje de error: ORA-01922:

Se debe especificar CASCADE para borrar USUARIO1 y no nos deja borrarlo, tenemos que añadir la opción CASCADE: **DROP USER** USUARIO1 CASCADE;

B. Privilegios

Un **privilegio** es la capacidad de un usuario dentro de la base de datos a realizar determinadas operaciones o a acceder a determinados objetos de otros usuarios. Ningún usuario puede llevar a cabo una operación si antes no se le ha concedido permiso.

Mediante la asignación de privilegios se permite o restringe el acceso a los datos o la realización de cambios en los datos, la posibilidad de realizar funciones del sistema, etcétera.

Cuando se crea un usuario es necesario darle privilegios para que pueda hacer algo. Oracle ofrece varios roles o funciones, tres de ellos se aplican al entorno de desarrollo: CONNECT, RESOURCE y DBA. Los demás están relacionados con la administración de la base de datos: EXP_FULL_DATABASE, IMP_FULL_DATABASE, DELETE_CATALOG_ROLE, DM_CATALOG_ROLE, HS_ADMIN_ROLE, AQ_USER_ROLE, etcétera.

Un rol o función está formado por un conjunto de privilegios. A continuación, se exponen los privilegios para cada uno de los roles que se aplican al entorno de desarrollo:

Roles (funciones)	Privilegios
CONNECT	ALTER SESSION, CREATE CLUSTER, CREATE DATABASE LINK, CREATE SEQUENCE, CREATE SESSION, CREATE SYNONYM, CREATE TABLE y CREATE VIEW.
RESOURCE	CREATE CLUSTER, CREATE INDEXTYPE, CREATE OPERATOR, CREATE PROCEDURE, CREATE TABLE, CREATE SEQUENCE, CREATE TRIGGER y CREATE TYPE
DBA	Posee todos los privilegios del sistema



Hay dos tipos de privilegios que podemos definir en la base de datos: privilegios sobre los objetos y privilegios del sistema.

- **Privilegios sobre los objetos**

  Estos privilegios nos permiten acceder y realizar cambios en los datos de los objetos de otros usuarios.

  Por ejemplo, el privilegio de consultar la tabla de otro usuario es un privilegio sobre objetos.

  Se dispone de los siguientes privilegios sobre los objetos tablas, vistas, secuencias y *procedures*:

Privilegio sobre los objetos	Tabla	Vista	Secuencia	Procedure
ALTER	X		X	
DELETE	X	X		
EXECUTE				X
INDEX	X			
INSERT	X	X		
REFERENCES	X			
SELECT	X	X	X	
UPDATE	X	X		

La orden para dar privilegios sobre los objetos es **GRANT,** con el siguiente formato:

```
GRANT {priv_objeto [,priv_objeto] ...|ALL [PRIVILEGES]}
      [(columna [,columna]...)]
ON    [usuario.]objeto}
TO    {usuario|rol|PUBLIC [,{usuario|rol|PUBLIC} ...]
[WITH GRANT OPTION];
```

Donde:

- ON especifica el objeto sobre el que se dan los privilegios.

- TO identifica a los usuarios o roles a los que se conceden los privilegios.

- ALL concede todos los privilegios sobre el objeto especificado.

- La cláusula WITH GRANT OPTION permite que el receptor del privilegio o rol se lo asigne a otros usuarios o roles.

- PUBLIC asigna los privilegios a todos los usuarios actuales y futuros. El propósito principal del grupo PUBLIC es garantizar el acceso a determinados objetos a todos los usuarios de la base de datos.

Con la orden GRANT se pueden conceder privilegios INSERT, UPDATE o REFERENCES sobre determinadas columnas de una tabla.



Caso práctico

5 **El usuario MILAGROS tiene una tabla de nombre TABLA1 que contiene la temperatura de una serie de ciudades. Concede a FRANCISCO los privilegios de SELECT e INSERT en TABLA1: `GRANT SELECT, INSERT ON TABLA1 TO FRANCISCO;` Ahora FRANCISCO puede acceder a la TABLA1 de MILAGROS de la siguiente manera: `SELECT * FROM MILAGROS.TABLA1;`**

El usuario MILAGROS da privilegios a JUAN para insertar en TABLA1: `GRANT INSERT ON TABLA1 TO JUAN;` Ahora Juan sólo puede insertar en la TABLA1 de MILAGROS, si intenta hacer SELECT se producirá un error:

```
SELECT * FROM MILAGROS.TABLA1;
          *
ERROR en línea 1:
ORA-01031: privilegios insuficientes
```

MILAGROS concede a FRANCISCO todos los privilegios sobre TABLA1: `GRANT ALL ON TABLA1 TO FRANCISCO;`

MILAGROS concede todos los privilegios sobre TABLA1 a todos los usuarios, incluyendo a los que se crean después de ejecutar esta orden: `GRANT ALL ON TABLA1 TO PUBLIC;`

MILAGROS concede a JUAN sobre TABLA1 para que pueda modificar sólo la columna temperatura: `GRANT UPDATE (TEMPERATURA) ON TABLA1 TO JUAN;` Si JUAN intenta modificar las dos columnas de la tabla, evidentemente, no podrá. En cambio, no tendrá ningún problema al modificar la columna a la cual tiene acceso: `UPDATE MILAGROS.TABLA1 SET TEMPERATURA=20 WHERE CIUDAD='MADRID';`

Con la opción **WITH GRANT OPTION** se puede dar al usuario que recibe los privilegios el privilegio para que él pueda concederlos a otros. Ahora MILAGROS concede a FRANCISCO el privilegio para insertar en TABLA1 y, además, para que él pueda pasar este privilegio a otros usuarios: `GRANT INSERT ON TABLA1 TO FRANCISCO WITH GRANT OPTION;`

Ahora FRANCISCO puede conceder el privilegio INSERT a otros usuarios sobre la tabla TABLA1 de MILAGROS. A continuación, da el privilegio INSERT sobre TABLA1 a JUAN: `GRANT INSERT ON MILAGROS.TABLA1 TO JUAN;`

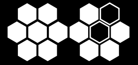


Actividades propuestas

1 **Concede el privilegio SELECT e INSERT sobre la tabla DEPART a uno de tus compañeros de clase con la opción de que se lo pueda conceder a otros.**

Concede el privilegio UPDATE sobre la columna APELLIDO de la tabla EMPLE a un compañero de clase.

Prueba los privilegios recibidos sobre las tablas.



- **Privilegios del sistema**

  Son los que dan derecho a ejecutar un tipo de comando SQL o a realizar alguna acción sobre objetos de un tipo especificado. Por ejemplo, el privilegio para crear *tablespaces* es un privilegio del sistema. Existen más de 80 tipos de privilegios distintos disponibles. Algunos de ellos son los siguientes:

PRIVILEGIO DEL SISTEMA	OPERACIONES AUTORIZADAS
	INDEX
CREATE ANY INDEX	Crear un índice en cualquier esquema, en cualquier tabla.
ALTER ANY INDEX	Modificar cualquier índice de la base de datos.
DROP ANY INDEX	Borrar cualquier índice de la base de datos.
	PRIVILEGE
GRANT ANY PRIVILEGE	Conceder cualquier privilegio de sistema.
	PROCEDURE
CREATE ANY PROCEDURE	Crear procedimientos almacenados, funciones y paquetes en cualquier esquema.
CREATE PROCEDURE	Crear procedimientos almacenados, funciones y paquetes en nuestro esquema.
ALTER ANY PROCEDURE	Modificar procedimientos almacenados, funciones y paquetes en cualquier esquema.
DROP ANY PROCEDURE	Borrar procedimientos almacenados, funciones y paquetes en cualquier esquema.
EXECUTE ANY PROCEDURE	Ejecutar procedimientos, funciones o referencias a paquetes públicos en cualquier esquema.
	PROFILE
CREATE PROFILE	Crear un perfil de usuario.
ALTER PROFILE	Modificar cualquier perfil.
DROP PROFILE	Borrar cualquier perfil.
	ROLE
CREATE ROLE	Crear roles.
ALTER ANY ROLE	Modificar roles.
DROP ANY ROLE	Borrar cualquier rol.
GRANT ANY ROLE	Dar permisos para cualquier rol de la base.
	SEQUENCE
CREATE SEQUENCE	Crear secuencias en nuestro esquema.
ALTER ANY SEQUENCE	Modificar cualquier secuencia de la base.
DROP ANY SEQUENCE	Borrar secuencias de cualquier esquema.
SELECT ANY SEQUENCE	Referenciar secuencias de cualquier esquema.
	SESSION
CREATE SESSION	Conectarnos a la base de datos.
ALTER SESSION	Manejar la orden ALTER SESSION.
RESTRICTED SESSION	Conectarnos a la base de datos cuando se ha levantado con STARTUP RESTRICT.
	SYNONYM
CREATE SYNONYM	Crear sinónimos en nuestro esquema.
CREATE PUBLIC SYNONYM	Crear sinónimos públicos.
DROP PUBLIC SYNONYM	Borrar sinónimos públicos.
CREATE ANY SYNONYM	Crear sinónimos en cualquier esquema.
DROP ANY SYNONYM	Borrar sinónimos de cualquier esquema.
	TABLE
CREATE TABLE	Crear tablas en nuestro esquema y generar índices sobre las tablas del esquema.
CREATE ANY TABLE	Crear una tabla en cualquier esquema.
ALTER ANY TABLE	Modificar una tabla en cualquier esquema.

(Continúa)

(Continuación)

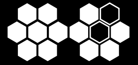
PRIVILEGIO DEL SISTEMA	OPERACIONES AUTORIZADAS
DROP ANY TABLE	Borrar una tabla en cualquier esquema.
LOCK ANY TABLE	Bloquear una tabla en cualquier esquema.
SELECT ANY TABLE	Hacer SELECT en cualquier tabla.
INSERT ANY TABLE	Insertar filas en cualquier tabla.
UPDATE ANY TABLE	Modificar filas en cualquier tabla.
DELETE ANY TABLE	Borrar filas de cualquier tabla.
	TABLESPACES
CREATE TABLESPACE	Crear espacios de tablas.
ALTER TABLESPACE	Modificar *tablespaces*.
MANAGE TABLESPACES	Poner *on-line* u *off-line* a cualquier *tablespace*.
DROP TABLESPACE	Eliminar tablespaces.
UNLIMITED TABLESPACE	Utilizar cualquier espacio de cualquier *tablespace*.
	TYPE
CREATE TYPE	Crea tipos de objeto y cuerpos de tipos de objeto en el propio esquema.
CREATE ANY TYPE	Crea tipos de objeto y cuerpos de tipos de objeto en cualquier esquema.
ALTER ANY TYPE	Modifica tipos de objeto en cualquier esquema.
DROP ANY TYPE	Elimina tipos de objeto y cuerpos de tipos de objeto en cualquier esquema.
EXECUTE ANY TYPE	Utiliza y hace referencia a tipos de objeto y tipos de colección en cualquier esquema.
UNDER ANY TYPE	Crea subtipos a partir de cualquier tipo de objeto no final.
	USER
CREATE USER	Crear usuarios y crear cuotas sobre cualquier espacio de tablas, establecer espacios de tablas por omisión y temporales.
ALTER USER	Modificar cualquier usuario. Este privilegio autoriza al que lo recibe a cambiar la contraseña de otro usuario, a cambiar cuotas sobre cualquier espacio de tablas, a establecer espacios de tablas por omisión, etcétera.
DROP USER	Eliminar usuarios.
	VIEW
CREATE VIEW	Crear vistas en el esquema propio.
CREATE ANY VIEW	Crear vistas en cualquier esquema.
DROP ANY VIEW	Borrar vistas en cualquier esquema.
	OTROS
SYSDBA	Ejecutar operaciones STARTUP y SHUTDOWN, ALTER DATABASE, CREATE DATABASE, ARCHIVELOG y RECOVERY, CREATE SPFILE
SYSOPER	Ejecutar operaciones STARTUP y SHUTDOWN, ALTER DATABASE, ARCHIVELOG y RECOVERY, CREATE SPFILE

El formato de la orden GRANT para asignar privilegios del sistema es:

```
GRANT {privilegio|rol} [, {privilegio|rol}, ....]
TO{usuario|rol|PUBLIC}[,{usuario|rol|PUBLIC}] ....
[WITH ADMIN OPTION];
```

Donde:

- TO identifica a los usuarios o roles a los que se conceden los privilegios.
- La cláusula WITH ADMIN OPTION permite que el receptor del privilegio o rol pueda conceder esos mismos privilegios a otros usuarios o roles.



Caso práctico



6 **Cuando creamos un usuario tenemos que darle privilegios para que, como mínimo, pueda iniciar sesión en la base de datos. Creamos el usuario PEDRO y le damos el privilegio de crear sesión (CREATE SESSION):** `CREATE USER PEDRO IDENTIFIED BY PEDRO QUOTA 500K ON USERS;`

```
GRANT CREATE SESSION TO PEDRO;
```

Se concede a PEDRO el rol CONNECT, lo que le permitirá tener todos los privilegios descritos para este rol (ALTER SESSION, CREATE CLUSTER, CREATE DATABASE LINK, CREATE SEQUENCE, CREATE SESSION, CREATE SYNONYM, CREATE TABLE y CREATE VIEW): `GRANT CONNECT TO PEDRO;`

Ahora se concede a PEDRO y a USUARIO1 el privilegio de administrador del sistema (DBA): `GRANT DBA TO PEDRO, USUARIO1;`

Para hacer que MILAGROS pueda borrar usuarios y, además, pueda conceder este privilegio a otros usuarios, se utiliza la opción WITH ADMIN OPTION: `GRANT DROP USER TO MILAGROS WITH ADMIN OPTION;`

Para hacer que todos los usuarios puedan hacer SELECT en cualquier tabla de cualquier usuario, escribimos: `GRANT SELECT ANY TABLE TO PUBLIC;`

- **Retirada de privilegios**

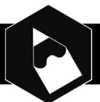
Al igual que se conceden privilegios, se pueden retirar. Para eso sirve la orden SQL REVOKE, que retira privilegios o roles concedidos a los usuarios y privilegios concedidos a los roles. El formato para retirar privilegios de objetos a los usuarios o roles es:

```
REVOKE {priv_objeto [,priv_objeto] ...|ALL [PRIVILEGES]}
ON [usuario.]objeto
FROM {usuario|rol|PUBLIC} [,{usuario|rol|PUBLIC}]...;
```

Y el formato para retirar privilegios de sistema o roles a usuarios o para retirar privilegios a roles es el siguiente:

```
REVOKE {priv_sistema|rol} [,{priv_sistema|rol}] ...
FROM {usuario|rol|PUBLIC} [,{usuario|rol|PUBLIC}]...;
```

Caso práctico



7 **MILAGROS retira los privilegios SELECT y UPDATE sobre TABLA1 a FRANCISCO:** `REVOKE SELECT, UPDATE ON TABLA1 FROM FRANCISCO;`

(Continúa)

(Continuación)

REVOKE ALL elimina todos los privilegios concedidos anteriormente sobre algún objeto. La opción WITH GRANT OPTION desaparece con el privilegio con el cual fue asignada. En este ejemplo, MILAGROS retira todos los privilegios concedidos a FRANCISCO y JUAN sobre TABLA1: `REVOKE ALL ON TABLA1 FROM FRANCISCO, JUAN;`

Retiramos el privilegio de borrar usuarios a MILAGROS: `REVOKE DROP USER FROM MILAGROS;`

Retiramos el privilegio de consultar cualquier tabla a todos los usuarios: `REVOKE SELECT ANY TABLE FROM PUBLIC;`

Retiramos el privilegio de administrador (DBA) a los usuarios JUAN y PEDRO: `REVOKE DBA FROM JUAN, PEDRO;`

- **Vistas con información de los privilegios**

  Para conocer los privilegios que han concedido o recibido los usuarios sobre los objetos o a nivel del sistema, podemos consultar las siguientes vistas del diccionario de datos:

  – `SESSION_PRIVS`: privilegios del usuario activo.

  – `USER_SYS_PRIVS`: privilegios de sistema asignados al usuario.

  – `DBA_SYS_PRIVS`: privilegios de sistema asignados a los usuarios o a los roles.

  – `USER_TAB_PRIVS`: concesiones sobre objetos que son propiedad del usuario, concedidos o recibidos por éste.

  – `USER_TAB_PRIVS_MADE`: concesiones sobre objetos que son propiedad del usuario (asignadas).

  – `USER_TAB_PRIVS_RECD`: concesiones sobre objetos que recibe el usuario.

  – `USER_COL_PRIVS`: concesiones sobre columnas en las que el usuario es el propietario, asigna el privilegio o lo recibe.

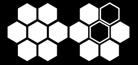
  – `USER_COL_PRIVS_MADE`: todas las concesiones sobre columnas de objetos que son propiedad del usuario.

  – `USER_COL_PRIVS_RECD`: concesiones sobre columnas recibidas por el usuario.



Actividades propuestas

2 **Escribe una secuencia de órdenes en la que se crea un usuario, se le asigna el privilegio de iniciar sesión en Oracle y de crear una tabla. Después conéctate con ese usuario y consulta los privilegios del sistema que tiene. Concede permiso SELECT sobre la tabla creada a otro usuario y consulta los privilegios sobre objetos concedidos y recibidos por él.**



C. Roles

Supongamos que un conjunto de usuarios del departamento de contabilidad requiere el mismo conjunto de privilegios para trabajar con ciertos datos. Este conjunto de privilegios se puede agrupar en un rol, de tal manera que es posible asignar el mismo rol a cada uno de los usuarios. Un **rol** o **función** es un conjunto de privilegios que recibe un nombre común para facilitar la tarea de asignación de éstos a los usuarios o a otros roles. Los privilegios de un rol pueden ser de sistema y a nivel de objeto. En primer lugar creamos el rol con la orden SQL CREATE ROLE y, a continuación, asignamos privilegios con la orden GRANT. El formato para crear un rol es:

```
CREATE ROLE NombreRol;
```

Para crear un rol se requiere el privilegio de sistema CREATE ROLE. Por ejemplo, creamos un rol llamado ACCESO: `CREATE ROLE ACCESO;`

Conceder privilegios a los roles

Una vez creado, hemos de concederle privilegios usando la orden GRANT. Por ejemplo, asignamos los privilegios al rol ACCESO: SELECT e INSERT sobre la tabla EMPLE, INSERT en la tabla DEPART, y CREATE SESSION para poder iniciar sesión en Oracle. El usuario que conceda estos privilegios ha de ser el propietario de las tablas EMPLE y DEPART (o debe tener autorización para conceder privilegios sobre estas tablas) y debe tener el privilegio CREATE SESSION con la posibilidad de concedérselo a otros usuarios:

```
GRANT SELECT, INSERT ON EMPLE TO ACCESO;

GRANT INSERT ON DEPART TO ACCESO;

GRANT CREATE SESSION TO ACCESO;
```

Para conceder el rol a un usuario escribimos: `GRANT ACCESO TO USUARIO;` El usuario podrá conectarse a la base de datos y hacer SELECT e INSERT en la tabla EMPLE, e INSERT en la tabla DEPART.

Se pueden añadir privilegios al rol ejecutando otra orden GRANT. Por ejemplo, añadimos el privilegio SELECT sobre la tabla DEPART al rol creado anteriormente: `GRANT SELECT ON DEPART TO ACCESO;`

Actividades propuestas



3 **Crea un usuario y concédele el rol creado (ACCESO). Añade el privilegio CREATE TABLE al rol. Consulta los privilegios de sistema que tiene asignados el usuario creado.**

- **Límites en privilegios sobre roles**

  Un rol puede decidir el acceso de un usuario a un objeto, pero no puede permitir la creación de objetos. Supongamos que el usuario creado anteriormente puede crear vistas (tiene el privilegio CREATE VIEW). También puede hacer SELECT en la tabla EMPLE, ya que tiene asignado el rol ACCESO. Pero no puede crear una vista sobre la tabla EMPLE debido a que recibió el privilegio SELECT a través del rol ACCESO.

- **Supresión de privilegios en los roles**

  La orden REVOKE permite suprimir los privilegios dados a los roles. Por ejemplo, para retirar del rol ACCESO el privilegio INSERT en la tabla EMPLE escribiremos lo siguiente: **REVOKE** INSERT ON EMPLE FROM ACCESO; Ahora retiramos del rol ACCESO el privilegio CREATE TABLE: **REVOKE** CREATE TABLE FROM ACCESO;

- **Supresión de un rol**

  La orden DROP ROLE permite eliminar un rol de la base de datos. Oracle retira el rol concedido a todos los usuarios y roles a los que se les concedió. Para poder eliminar un rol es necesario ser administrador o tener el privilegio DROP ANY ROLE. Éste es el formato:

  DROP ROLE *NombreRol*;

  Por ejemplo, para eliminar el rol ACCESO escribimos: **DROP ROLE** ACCESO;

- **Establecer un rol por defecto**

  Es posible establecer un rol por defecto a un usuario mediante la orden ALTER USER, antes tiene que ser concedido el rol al usuario. El formato es:

  ALTER USER *Nombreusuario* **DEFAULT ROLE** *nombre_rol*;

  Por ejemplo, establecemos un rol por defecto (ACCESO) para el usuario creado anteriormente: **ALTER USER** USUARIO DEFAULT ROLE ACCESO;

Con CREATE USER no se puede asignar un rol por defecto.

- **Información sobre roles en el diccionario de datos**

  Para saber a qué usuarios se ha concedido acceso a un rol, o los privilegios que se han concedido a un rol, se pueden consultar las siguientes vistas del diccionario de datos:

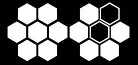
  – ROLE_SYS_PRIVS: privilegios del sistema asignados a roles.

  – ROLE_TAB_PRIVS: privilegios sobre tablas aplicados a roles.

  – ROLE_ROLE_PRIVS: roles asignados a otros roles.

  – SESSION_ROLES: roles activos para el usuario.

  – USER_ROLE_PRIVS: roles asignados al usuario.



D. Perfiles

Un **perfil** es un conjunto de límites a los recursos de la base de datos. Se pueden utilizar perfiles para poner límites a la cantidad de recursos del sistema y de la base de datos disponibles para un usuario y para gestionar las restricciones de contraseña.

Si no se crean perfiles en una base de datos, entonces se utiliza el perfil por defecto (DEFAULT) que especifica recursos ilimitados para todos los usuarios.

La sentencia SQL para crear un perfil es CREATE PROFILE, cuyo formato es:

```
CREATE PROFILE nombreperfil LIMIT
    {parámetros_recursos | parámetros_contraseña}
     {Entero [K|M] | UNLIMITED | DEFAULT};
```

Donde:

- parámetros_recursos: SESSIONS_PER_USER, CPU_PER_SESSION, CPU_PER_CALL, CONNECT_TIME, IDLE_TIME, LOGICAL_READS_PER_SESSION, LOGICAL_READS_PER_CALL, PRIVATE_SGA, COMPOSITE_LIMIT

- parámetros_contraseña: FAILED_LOGIN_ATTEMPTS, PASSWORD_LIFE_TIME, PASSWORD_REUSE_TIME, PASSWORD_REUSE_MAX, PASSWORD_LOCK_TIME, PASSWORD_GRACE_TIME, PASSWORD_VERIFY_FUNCTION

- UNLIMITED significa que no hay límite sobre un recurso particular.

- DEFAULT coge el límite del perfil DEFAULT.

Donde los recursos son los que aparecen en la tabla siguiente:

RECURSO	FUNCIÓN
SESSIONS_PER_USER	Número de sesiones múltiples concurrentes permitidas por nombre de usuario.
CONNECT_TIME	Indica el número de minutos que puede estar una sesión conectada.
IDLE_TIME	Indica el número de minutos que puede estar una sesión conectada sin ser utilizada de forma activa.
CPU_PER_SESSION	Limita el tiempo máximo de CPU por sesión. Este valor se expresa en centésimas de segundo.
CPU_PER_CALL	Limita el tiempo máximo de CPU por llamada (de análisis, ejecución o búsqueda). Se expresa en centésimas de segundo.
LOGICAL_READS_PER_SESSION	Limita el número de bloques de datos leídos en una sesión.
LOGICAL_READS_PER_CALL	Limita el número de bloques de datos leídos por llamada (de análisis, ejecución o búsqueda).

(Continúa)

(Continuación)

RECURSO	FUNCIÓN
PRIVATE_SGA	Indica la cantidad de espacio privado que una sesión puede reservar en el área SQL compartida de la SGA (para la opción servidor compartido).
COMPOSITE_LIMIT	Indica un límite compuesto basado en los límites anteriores.
FAILED_LOGIN_ATTEMPTS	Número de intentos de acceso sin éxito consecutivos que producirá el bloqueo de la cuenta.
PASSWORD_LIFE_TIME	Número de días que puede utilizarse una contraseña antes de que caduque.
PASSWORD_REUSE_TIME	Número de días que deben pasar antes de que se pueda reutilizar una contraseña.
PASSWORD_REUSE_MAX	Número de veces que debe cambiarse una contraseña antes de poder reutilizarla.
PASSWORD_LOCK_TIME	Número de días que quedará bloqueada una cuenta si se sobrepasa el valor del parámetro FAILED_LOGIN_ATTEMPTS.
PASSWORD_GRACE_TIME	La duración en días del periodo de gracia durante el cual una contraseña puede cambiarse cuando ha alcanzado su valor PASSWORD_LIFE_TIME.

Para activar el uso de perfiles en el sistema, el administrador ha de ejecutar esta orden: **ALTER SYSTEM** SET RESOURCE_LIMIT=TRUE; (FALSE desactiva la utilización de perfiles). Para asignar un perfil a un usuario se puede utilizar la orden ALTER USER: ALTER USER USUARIO PROFILE nombreperfil; O bien al crear el usuario se le puede asignar un perfil.



Caso práctico

8 **Por razones de seguridad, creamos el perfil PERFIL1, en el que limitamos a uno el número de sesiones concurrentes por usuario y a dos minutos el tiempo de conexión permitido por sesión:**

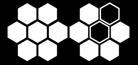
```
CREATE PROFILE PERFIL1 LIMIT SESSIONS_PER_USER 1 CONNECT_TIME 2;
```

Dado que los demás límites de recurso no se mencionan en la instrucción CREATE PROFILE, se utilizarán los valores asignados por defecto por el sistema.

A continuación, se crea un usuario llamado PRUEBA. Se le asigna este perfil y se le concede el rol CONNECT:

```
CREATE USER PRUEBA IDENTIFIED BY PRUEBAQUOTA 100K ON USERS PROFILE PERFIL1;
GRANT CONNECT TO PRUEBA;
```

(Continúa)



(Continuación)

Ejecutamos SQL*Plus una vez con el usuario PRUEBA y se conecta perfectamente. Si volvemos a ejecutar SQL*Plus (sin cerrar la sesión anterior) con el usuario PRUEBA, nos aparecerá este mensaje de error:

```
ERROR:
ORA-02391: exceeded simultaneous SESSIONS_PER_USER limit
```

Esto se debe a que PRUEBA tiene limitado el número de sesiones concurrentes a una.

Al cabo de un tiempo conectado, el usuario PRUEBA intenta acceder a una tabla, y Oracle visualiza un mensaje de error, indicando que se ha pasado el tiempo de conexión que tenía asignado:

```
ERROR:
ORA-02399: ha excedido el tiempo máximo de conexión, desconectando
```

A continuación, se crea un perfil en el que se permiten 3 intentos de acceso fallidos para la cuenta, el cuarto producirá el bloqueo de la cuenta: **CREATE PROFILE** PERFIL2 LIMIT FAILED_LOGIN_ATTEMPTS 3;

Se asigna el perfil al usuario PRUEBA: ALTER USER PRUEBA PROFILE PERFIL2; Si hay 3 conexiones fallidas consecutivas en la cuenta PRUEBA (por ejemplo, se ha escrito mal la contraseña) quedará automáticamente bloqueada por Oracle, aunque se conecte con la contraseña correcta:

```
SQL> CONNECT PRUEBA/PRUEBA
ERROR:
ORA-28000: the account is locked
```

Para desbloquear una cuenta el administrador de la base de datos tiene que ejecutar: **ALTER USER** PRUEBA ACCOUNT UNLOCK;

A continuación, se modifica el perfil anterior que obligará a los usuarios que lo tengan asignado a cambiar su contraseña cada 2 días:

ALTER PROFILE PERFIL2 LIMIT PASSWORD_LIFE_TIME 2;

La vista DBA_PROFILES contiene información sobre los límites.

La orden ALTER PROFILE permite modificar una determinada configuración de perfil. El formato es el mismo que el de la orden CREATE PROFILE.

Borrado de un perfil

Para borrar un perfil de la base de datos se usa la orden DROP PROFILE, que tiene el siguiente formato:

DROP PROFILE *NombrePerfil* [CASCADE];

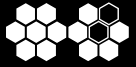
Si algún usuario lo tiene asignado es necesario incluir la opción CASCADE. Ejemplo: **DROP PROFILE** PERFIL2 CASCADE;

Conceptos básicos



A continuación, se muestra un resumen de las órdenes vistas en la Unidad:

USUARIOS	CREATE USER DROP USER ALTER USER
PRIVILEGIOS	GRANT REVOKE
ROLES	CREATE ROLE DROP ROLE ALTER ROLE
PERFILES	CREATE PROFILE DROP PROFILE ALTER PROFILE
TABLESPACES	CREATE TABLESPACE DROP TABLESPACE ALTER TABLESPACE
SECUENCIAS	CREATE SEQUENCE DROP SEQUENCE ALTER SEQUENCE
ÍNDICES	CREATE INDEX DROP INDEX ALTER INDEX
DATABASE LINKS	CREATE DATABASE LINK DROP DATABASE LINK ALTER DATABASE LINK



Actividades complementarias



1 **Crea una base de datos de forma manual llamada ORACLE10. Sigue el procedimiento del Caso práctico 2.**

2 **Una vez creada la base de datos, crea un usuario y dale permiso para conectarse y crear objetos. Crea una tabla con el formato que desees. Resuelve las situaciones que se presenten.**

3 **Crea un rol que tenga los siguientes privilegios: INSERT y SELECT en DEPART y EMPLE, CREATE SESSION, CREATE TYPE, CREATE TABLE y CREATE VIEW.**

4 **Crea un usuario llamado COMPRADOR. El *tablespace*, por defecto, es COMPRAS. Se le asigna 1 Megabyte en el *tablespace* COMPRAS. El *tablespace* temporal será TEMP. Se le asigna el rol anterior.**

5 **Realiza la siguiente secuencia de instrucciones en el orden indicado:**

1. Crea un usuario de base de datos que tenga funciones de administrador.
2. Conéctate con el nombre de usuario creado.
3. Crea varias tablas en el propio esquema.
4. Crea cinco usuarios nuevos asignándoles un *tablespace* por defecto y cuota (USU1, USU2, USU3, USU4 y USU5).
5. Da permiso a uno de los usuarios (USU1) sólo para que pueda conectarse a la base de datos.
6. Crea un rol que permita conectarse a la base de datos y hacer SELECT sobre algunas tablas.
7. Concede el rol creado a dos de los usuarios creados anteriormente (USU2 y USU3).
8. Concede al usuario USU4 privilegios sobre algunas tablas con la opción de poder concedérselos a otros usuarios.
9. Concede al usuario USU5 cuatro privilegios de sistema, dos de ellos, con la opción de poder concedérselos a otros usuarios.
10. Concede a todos los usuarios de la base de datos privilegios para que puedan modificar ciertas columnas de algunas tablas.
11. Quita a los usuarios USU3 y USU4 todos los privilegios que tenían asignados.
12. Haz que USU5 sólo pueda conectarse en dos sesiones concurrentes a la vez.
13. Limita el tiempo de conexión a la base de datos a cinco minutos a los usuarios USU2 y USU3.

6 **Indica cuál de las siguientes afirmaciones es correcta:**

a) A nivel lógico, una Base de Datos se divide en espacios de tabla (*tablespaces*), segmentos, extensiones y bloques de Oracle.

b) Un segmento es una agrupación de bloques de Oracle contiguos.

c) Los archivos que componen una base de datos constituyen una división lógica de la misma.

d) Un *tablespace* se almacena en un único archivo físico.

7 **Después de crear el usuario pepe:**

```
CREATE USER PEPE IDENTIFIED BY PEPE
DEFAULT TABLESPACE DATOS
QUOTA 1M ON DATOS2;
```

Indica la o las afirmaciones correctas:

a) Pepe puede crear objetos en el *tablespace* DATOS.

b) Pepe puede crear objetos en el *tablespace* DATOS2.

c) Pepe no puede crear objetos en el *tablespace* DATOS porque no tiene espacio asignado en dicho *tablespace*.

d) Pepe puede crear objetos en los dos *tablespaces*.

Administración de Oracle II

13

En esta unidad aprenderás a:

1. Crear y gestionar los tablespaces.
2. Utilizar secuencias e índices.
3. Crear copias de seguridad.
4. Restaurar una base de datos.
5. Analizar los registros de deshacer o Redo Log.
6. Realizar auditorías a objetos de bases de datos.



13.1 Gestión de tablespaces

Ya se ha explicado que una base de datos está formada por un conjunto de archivos de datos, pero ¿cómo agrupa Oracle estos archivos? Lo hace usando un objeto denominado *tablespace* o *espacio de tablas*. Se llama así porque contiene tablas de datos. Antes de introducir datos en la base de datos, es necesario crear un *tablespace* y, seguidamente, las tablas en las que se van a introducir los datos. Estas tablas se deben almacenar en un *tablespace*.

Un **tablespace** es una unidad lógica de almacenamiento de datos representada físicamente por uno o más archivos de datos. Se recomienda no mezclar datos de diferentes aplicaciones en el mismo *tablespace*, es decir, se debe crear un *tablespace* para almacenar los datos de la aplicación de gestión de almacén, otro *tablespace* para almacenar los datos de la nómina de los empleados, etcétera. Al instalar Oracle se crean varios: SYSTEM, USERS, TEMP o UNDOTBS1. Veamos a continuación cómo se crean y gestionan los *tablespaces*.

A. Creación de un tablespace

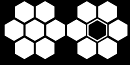
Para crear un *tablespace* se usa la orden CREATE TABLESPACE que permite asignar uno o más archivos al espacio de tablas y especificar un espacio por omisión para cualquiera de las tablas creadas sin un espacio de tabla explícitamente mencionado en una sentencia CREATE TABLE. El formato es:

```
CREATE [UNDO] TABLESPACE nombretablespace
DATAFILE  'nombrearchivo' [SIZE entero[K|M]][REUSE]
                  [AUTOEXTEND {OFF|ON cláusulas}]
         [,'nombrearchivo' [SIZE entero[K|M]][REUSE]
[AUTOEXTEND {OFF|ON cláusulas}] ]...
 [DEFAULT STORAGE
(
          INITIAL  tamaño
          NEXT  tamaño
          MINEXTENTS  tamaño
          MAXEXTENTS  tamaño
          PCTINCREASE valor
)]
[ONLINE|OFFLINE]
[PERMANENT|TEMPORARY]
[EXTENT MANAGEMENT
  {DICTIONARY|LOCAL {AUTOALLOCATE |UNIFORM [SIZE tamaño
[K|M]]}}];
```

Un **bloque de datos** es la unidad de acceso a disco de Oracle.
Una **extensión** es un conjunto de bloques de datos contiguos.
Un **segmento** es un conjunto de extensiones.

Donde:

- UNDO especifica que se crea un *tablespace* de tipo «deshacer cambios», es decir, de Rollback. Si no hay un *tablespace* de deshacer se crea uno con la opción UNDO y Oracle lo asigna automáticamente a la instancia como el *tablespace* de deshacer. En éste no se podrán crear objetos. Está reservado para las acciones de deshacer cambios.



- DATAFILE especifica el archivo o archivos de datos de que constará el *tablespace*.

- SIZE entero especifica el tamaño del *tablespace*, que puede venir dado en Kilobytes (K) o en Megabytes (M). Si ponemos K, se multiplica el entero por 1.024. Si ponemos M, se multiplica por 1.048.576.

- REUSE reutiliza el archivo si ya existe, o lo crea si no existe.

- DEFAULT STORAGE define el almacenamiento por omisión para todos los objetos que se creen en este espacio de tabla. Fija la cantidad de espacio si no se especifica en la sentencia CREATE TABLE.

- INITIAL extensión inicial. Especifica el tamaño en bytes de la primera extensión del objeto. El tamaño se puede especificar en Kilobytes (K) o Megabytes (M).

- NEXT extensión siguiente. Especifica el tamaño de la siguiente extensión que se va a asignar al objeto. También se puede especificar K o M. El valor por defecto es el tamaño de un bloque de datos (el tamaño del bloque se especifica en un parámetro de inicio de Oracle: *db_block_size* y debe ser múltiplo del tamaño del bloque del sistema operativo del servidor).

- MINEXTENTS reserva extensiones adicionales más allá de la extensión inicial que se da a la tabla por omisión. Este parámetro permite asignar una gran cantidad de espacio cuando se crea un objeto, incluso si el espacio disponible no está contiguo. El valor por omisión es 1, que significa que Oracle sólo asigna la extensión inicial. Si el valor es mayor que 1, Oracle calcula el tamaño de las extensiones subsiguientes basándose en los valores de los parámetros INITIAL, NEXT y PCTINCREASE.

- MAXEXTENTS es el número total de extensiones, incluida la primera, que Oracle puede asignar al objeto. El valor depende del tamaño del bloque de datos.

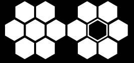
- PCTINCREASE es un factor de crecimiento para la extensión. El valor por omisión es 50, lo que significa que cada extensión subsiguiente será un 50 por 100 más grande que la extensión anterior. El valor de la siguiente extensión es: NEXT=NEXT + (PCTINCREASE*NEXT)/100

- ONLINE, OFFLINE. Con ONLINE el *tablespace* está disponible después de crearlo, es el valor por defecto. OFFLINE impide su acceso.

- PERMANENT, TEMPORARY. PERMANENT es la opción por defecto. TEMPORARY indica que el *tablespace* solo podrá albergar objetos temporales. Por ejemplo, segmentos que se utilizan en ordenaciones de datos.

- AUTOEXTEND cláusulas: activa o desactiva el crecimiento automático de los archivos de datos del *tablespace*. Cuando un *tablespace* se llena podemos usar esta opción para que el tamaño del archivo o archivos de datos asociados crezca automáticamente. AUTOEXTEND OFF desactiva el crecimiento automático. El formato AUTOEXTEND ON es:

  ```
  AUTOEXTEND ON NEXT entero {K|M} MAXSIZE {UNLIMITED|entero
  {K|M}}
  ```



`NEXT entero`: es el incremento de espacio en disco expresado, en Kilobytes o en Megabytes, que se reservará automáticamente para el archivo.

`MAXSIZE`: es el máximo espacio en disco reservado para la extensión automática del archivo.

`UNLIMITED`: significa que no hay límite del espacio en disco reservado.

- `EXTENT MANAGEMENT` especifica la gestión de las extensiones del *tablespace*:

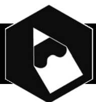
  `DICTIONARY` especifica que se gestiona mediante tablas de diccionario. Es la opción por defecto si el parámetro COMPATIBLE es inferior a 9.0.0.

  `LOCAL` especifica que se gestiona localmente mediante un mapa de bits.

  `AUTOALLOCATE`: el *tablespace* lo gestiona el sistema. Los usuarios no podrán especificar el tamaño de una extensión. Es la opción por defecto si el parámetro COMPATIBLE es 9.0.0 o superior.

  `UNIFORM` especifica que el *tablespace* se gestiona con extensiones uniformes de bytes SIZE. El tamaño SIZE por defecto es 1 MB. Si se especifica LOCAL no se puede especificar DEFAULT, MINIMUM EXTENT o TEMPORARY.

Caso práctico



1 A continuación, se crea un *tablespace* de 15 Megabytes llamado TRABAJO. El tamaño inicial para el objeto que se cree en el *tablespace* (por ejemplo, una tabla) es de 10 K. El tamaño de la siguiente extensión del objeto también es 10 K; cada extensión subsiguiente será un 25 por 100 más grande que la anterior. Asignamos dos archivos a este *tablespace* 'TRABAJO1.ORA', de 10M, y 'TRABAJO2.ORA', de 5M: (si no ponemos la ubicación el *tablespace* lo creará en C:\WINNT\SYSTEM32\):

```
CREATE TABLESPACE TRABAJO DATAFILE 'TRABAJO1.ORA' SIZE 10M,
'TRABAJO2.ORA' SIZE 5M
DEFAULT STORAGE (INITIAL 10K NEXT 10K PCTINCREASE 25);
```

Se crea un *tablespace* de 100K llamado PEQUE. Asignamos el archivo 'PEQUE.ORA', habilitando el crecimiento automático de 120K para la extensión siguiente dentro de un espacio máximo de 1M:

```
CREATE TABLESPACE PEQUE DATAFILE 'PEQUE.ORA' SIZE 100K
AUTOEXTEND ON NEXT 120K MAXSIZE 1M;
```

Se crea un *tablespace* de deshacer de 10M llamado DESHACER. Asignamos el archivo 'DESHACER.ORA', habilitando el crecimiento automático de 512K para la extensión siguiente dentro de un espacio máximo de ilimitado:

```
CREATE UNDO TABLESPACE DESHACER DATAFILE 'DESHACER.ORA' SIZE 10M REUSE AUTOEX-
TEND ON NEXT 512K MAXSIZE UNLIMITED;
```

- ***Tablespaces* temporales**
  Aunque la creación de un *tablespace* temporal se puede hacer con el comando anterior, es aconsejable utilizar la orden CREATE TEMPORARY TABLESPACE. El formato es el que sigue:

```
CREATE TEMPORARY TABLESPACE nombretablespace
TEMPFILE  'nombrearchivo' [SIZE entero[K|M]][REUSE]
                    [AUTOEXTEND {OFF|ON cláusulas}]
          [,'nombrearchivo' [SIZE entero[K|M]][REUSE]
   [AUTOEXTEND {OFF|ON cláusulas}] ]...
          [MAXSIZE {UNLIMITED | entero {K|M}}]
          [EXTENT MANAGEMENT LOCAL UNIFORM [SIZE entero
{K|M}]];
```

  TEMPFILE se puede omitir si se ha definido el parámetro de inicialización DB_CREATE_FILE_DEST, en el que Oracle crea un archivo temporal de 100 MB gestionado por él mismo.

  El tamaño de las extensiones SIZE serán múltiplos del parámetro SORT_AREA_SIZE. El segmento temporal recibirá bloques del tamaño que aquí pongamos. Por ejemplo, la orden que se muestra crea un *tablespace* temporal de 10 M llamado TEMPEJER y se ubicará en el TEMPFILE con el nombre indicado:

```
CREATE TEMPORARY TABLESPACE TEMPEJER TEMPFILE 'C:\ORA-
CLE\ PRODUCT\10.1.0\ORADATA\ORCL\TEMP_EJERCI.DBF' SIZE
10M;
```

- **Vistas con información sobre tablespaces**
  Existen varias vistas para obtener información sobre *tablespaces*. Algunas de ellas son:

  - DBA_DATA_FILES: Muestra información sobre los archivos utilizados por los *tablespaces* (FILE_NAME). El espacio está definido en bytes (BYTES). El máximo tamaño que puede llegar a tener (MAXBYTES), etc. Para consultarla es necesario que el usuario SYS dé privilegios.

  - USER_FREE_SPACE: Muestra las extensiones libres en *tablespaces* a las que puede acceder el usuario. No tienen por qué estar en bloques consecutivos. Por ejemplo, conectamos como usuario PRUEBA y consultamos las extensiones libres. Las columnas para esta vista son:

    `FILE_ID`: Nº. de identificación del archivo.

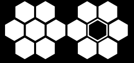
    `BYTES`: Es el número bytes libres.

    `BLOCK_ID`: Identificación del primer bloque libre.

    `BLOCKS`: Nº. de bloques libres.

    `RELATIVE_FNO`: Número relativo del fichero en la primera extensión del bloque.

  - DBA_FREE_SPACE: Muestra extensiones libres en todos los *tablespaces*.



– DBA_TABLESPACES: Muestra la descripción de todos los *tablespaces*. La orden siguiente visualiza los nombres de los *tablespaces* y el tipo:

```
SQL> SELECT TABLESPACE_NAME, CONTENTS FROM DBA_TABLES-
PACES;
TABLESPACE_NAME                   CONTENTS
------------------------------    ---------
SYSTEM                            PERMANENT
UNDOTBS1                          UNDO
SYSAUX                            PERMANENT
TEMP                              TEMPORARY
USERS                             PERMANENT
EXAMPLE                           PERMANENT
TRABAJO                           PERMANENT
PEQUE                             PERMANENT
DESHACER                          UNDO
TEMPEJER                          TEMPORARY
```

– DBA_TS_QUOTAS: Muestra los bytes utilizados por los usuarios en cada *tablespace*. Para consultar estas vistas es necesario que el usuario SYS nos dé privilegios. Las columnas para esta vista son:

`BYTES`: Es el número de bytes usados por el usuario.

`MAX_BYTES`: Nº. máximo de bytes que tiene el usuario asignado. Si el valor es –1 significa que tiene asignado un Nº. de bytes ilimitado.

`BLOCKS`: Nº. de bloques usados.

`MAX_BLOCKS`: Máximo número de bloques.

Actividades propuestas



1 **A partir de las vistas DBA_DATA_FILES y DBA_TS_QUOTAS, haz una consulta que contenga la siguiente información: Nombre del tablespace, Nombre Usuario, Tamaño total del tablespace, bytes usados por el usuario.**

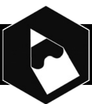
B. Modificación de tablespaces

Los *tablespaces*, una vez creados, se pueden modificar, es decir, es posible añadir nuevos archivos a un *tablespace* existente, modificar las cláusulas de almacenamiento para los objetos que se almacenen en el *tablespace*, activarlo y desactivarlo, etcétera. La modificación se lleva a cabo con la orden ALTER TABLESPACE, cuyo formato es:

```
ALTER TABLESPACE nombretablespace
{
[ADD DATAFILE 'nombrearchivo' [SIZE entero[K|M]][REUSE]
                    [AUTOEXTEND ON ..|OFF]
              [,'nombrearchivo' [SIZE ntero[K|M]][REUSE]
          [AUTOEXTEND ON ..|OFF]]..
]
 [RENAME DATAFILE 'archivo' [,'archivo']...
                 TO 'archivo' [,'archivo'] ]
 [DEFAULT STORAGE ClausulasAlmacenamiento ]
 [ONLINE|OFFLINE]
};
```

Donde:

- `nombretablespace` es el nombre del *tablespace* que se quiere modificar.
- `ADD_DATAFILE` añade al *tablespace* uno o varios archivos.
- `RENAME DATAFILE` cambia el nombre de un archivo existente del *tablespace*. Este cambio se tiene que hacer desde el sistema operativo y, después, ejecutar la orden SQL. El *tablespace* debe estar desactivado (*offline*) mientras se produce el cambio.
- `DEFAULT STORAGE` especifica los nuevos parámetros de almacenamiento para todos los objetos que se creen a partir de ahora en este *tablespace*.
- `ONLINE` pone el espacio de tablas en línea (activado).
- `OFFLINE` pone el espacio de tablas fuera de línea (desactivado).



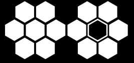
Caso práctico

2 Se agrega un archivo al *tablespace* TRABAJO de 6 Megabytes llamado 'TRABAJO3.ORA'

```
ALTER TABLESPACE  TRABAJO ADD DATAFILE 'TRABAJO3.ORA' SIZE 6M;
```

Renombramos los archivos TRABAJO1.ORA y TRABAJO2.ORA del *tablespace* TRABAJO se llamarán ahora TRABA1.ORA y TRABA2.ORA, respectivamente. Los pasos para renombrar archivos asociados a un *tablespace* son los siguientes:

1. Desactivar el *tablespace* TRABAJO: `ALTER TABLESPACE TRABAJO OFFLINE;`
2. Copiar los archivos TRABAJO1.ORA y TRABAJO2.ORA a TRABA1.ORA y TRABA2.ORA utilizando los comandos del sistema operativo.
3. Usar la orden ALTER TABLESPACE con la opción RENAME DATAFILE: `ALTER TABLESPACE  TRABAJO RENAME DATAFILE 'TRABAJO1.ORA', 'TRABAJO2.ORA' TO 'TRABA1.ORA','TRABA2.ORA';`
4. Activar el *tablespace* TRABAJO: `ALTER TABLESPACE TRABAJO ONLINE;`



Actividades propuestas



2 **Vamos a probar la opción de crecimiento automático para los archivos de un *tablespace*. Partimos del *tablespace* PEQUE creado anteriormente. Crea una tabla en este *tablespace* e inserta filas hasta que Oracle devuelva un error (porque se ha llenado el *tablespace*).**

- Modifica entonces el *tablespace* PEQUE añadiendo un archivo llamado 'MAYOR.ORA' de 100 K, que se va extendiendo automáticamente hasta 200 K cuando se llena. Como máximo, el espacio utilizado en disco para este archivo será de 1 MB.

- Sigue insertando filas hasta que sobrepase 1 MB de espacio en disco. Oracle visualizará un mensaje de error similar al anterior. Modifícalo de nuevo añadiendo un archivo de 100 K, que se va extendiendo automáticamente a 200 K sin indicar máximo tamaño. Prueba de nuevo a insertar filas.

C. Borrado de tablespaces

Para borrar un *tablespace* que ya no utilizamos se emplea DROP TABLESPACE.

Su formato es:

```
DROP TABLESPACE nombretablespace
[INCLUDING CONTENTS [AND DATAFILES] [CASCADE CONSTRAINTS]];
```

Donde:

- nombretablespace es el nombre del *tablespace* que se va a suprimir.

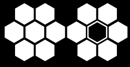
- La opción INCLUDING CONTENTS permite borrar un *tablespace* que tenga datos. Sin esta opción, únicamente se puede suprimir un *tablespace* vacío.

- AND DATAFILES borra los archivos de datos asociados y CASCADE CONSTRAINTS borra las relaciones de integridad referencial que afecten a las tablas del *tablespace* suprimido.

Se recomienda poner el *tablespace* OFFLINE antes de borrarlo para asegurarnos de que no haya sentencias SQL que estén accediendo a datos del *tablespace*, en cuyo caso no sería posible borrarlo. Por ejemplo, para borrar el *tablespace* PEQUE y los archivos de datos asociados escribiremos:

```
DROP TABLESPACE PEQUE INCLUDING CONTENTS AND DATAFILES;
```

Las órdenes CREATE TABLE y CREATE TABLESPACE contienen más cláusulas. En el tema sólo se han visto algunas. Para más información consultar el manual de referencia de Oracle.



D. Parámetros de almacenamiento

Hasta ahora, en las tablas que hemos creado no se ha definido ningún parámetro de almacenamiento, pues éste venía definido en la cláusula DEFAULT STORAGE del *tablespace* que contiene a la tabla. No obstante, en la orden CREATE TABLE podemos especificar los parámetros de almacenamiento para la tabla cuando tengamos que asignar parámetros distintos a los que asigna el *tablespace*.

El formato de CREATE TABLE, usando estos parámetros, es el siguiente:

```
CREATE  TABLE  Nombretabla
(
  Columna1   Tipo_dato  [NOT NULL],
  Columna2   Tipo_dato  [NOT NULL],
  [restricciones_de_tabla]............
)
STORAGE
   (
    INITIAL  tamaño
    NEXT  tamaño
    MINEXTENTS  tamaño
    MAXEXTENTS  tamaño
    PCTINCREASE valor
    )
[TABLESPACE nombretablespace];
```

El significado de los parámetros de la cláusula STORAGE es el mismo que el de los parámetros usados en la orden CREATE TABLESPACE.

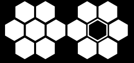
E. Desconexión de un tablespace

Con la base de datos abierta se pueden desconectar los *tablespaces*, excepto SYSTEM o cualquier *tablespace* con algún segmento de Rollback activo o segmentos temporales. Cuando se desconecta un *tablespace* el servidor Oracle desconecta los archivos asociados. Los usuarios que intenten acceder a objetos de un *tablespace* desconectado obtendrán un mensaje de error. El formato es el siguiente:

```
ALTER TABLESPACE nombretablespace { ONLINE |
 OFFLINE [NORMAL|TEMPORARY|IMMEDIATE]};
```

OFFLINE. Desactiva el *tablespace* e impide el acceso a sus segmentos. En modo NORMAL limpia la SGA de todos los bloques de los archivos de datos de este *tablespace*. Es la opción predeterminada. No es necesario recuperarlo antes de volver a activarlo.

Con TEMPORARY, Oracle ejecuta un *checkpoint* para todos los archivos de datos activos del *tablespace*. Antes de activarlo puede requerir recuperación. IMMEDIATE no ejecuta *checkpoint* y no asegura la disponibilidad de los archivos de datos asociados. Antes de activarlo requiere recuperación física.



13.2 Otros objetos

En este apartado se estudian otros objetos que se han nombrado a lo largo de esta Unidad o en las anteriores y que pueden ser de bastante utilidad: las *secuencias* y los *índices*.

A. Secuencias

Una **secuencia** es un objeto de base de datos que sirve para generar números enteros únicos. Es muy útil para generar automáticamente valores para claves primarias. Para crear una secuencia en el esquema propio es necesario tener el privilegio CREATE SEQUENCE. Se crea una secuencia en cualquier otro esquema con el privilegio CREATE ANY SEQUENCE. El formato para crear una secuencia es éste:

Los valores por defecto son: INCREMENT BY 1, NOMINVALUE, NOMAXVALUE y NOORDER.

```
CREATE SEQUENCE nombresecuencia
[INCREMENT BY entero]
[START WITH entero]
[MAXVALUE entero |NOMAXVALUE]
[MINVALUE entero |NOMINVALUE]
[CYCLE|NOCYCLE]
[ORDER|NOORDER]
[CACHE entero |NOCACHE];
```

Donde:

- `INCREMENT BY entero` especifica el intervalo de crecimiento de la secuencia. Si se omite, se asume valor 1. Si es negativo, produce un decremento de la secuencia.
- `START WITH entero` es el número con el que comienza la secuencia.
- `MAXVALUE entero` es el número más alto que generará la secuencia. Este entero debe ser menor o igual que el entero especificado en START WITH y mayor que el entero especificado en MINVALUE.
- `NOMAXVALUE` señala que el valor máximo para una secuencia ascendente es 10^{27} y para una secuencia descendente –1.
- `MINVALUE entero` es el número más bajo que generará la secuencia. El entero debe ser menor o igual que el entero especificado en START WITH y menor que el entero especificado en MAXVALUE.
- `NOMINVALUE` indica que el valor mínimo para una secuencia ascendente es 1 y -10^{26} para una secuencia descendente.
- `CYCLE|NOCYCLE`. `CYCLE` reanuda la secuencia cuando llega al máximo o al mínimo valor; `NOCYCLE` no la reanuda.
- `ORDER|NOORDER`. `ORDER` garantiza que los números de secuencia se generan en el orden requerido; `NOORDER` no lo garantiza. Si se omiten ambas, se asume NOORDER. En modo exclusivo, las secuencias siempre se generan en orden ascendente.

- CACHE entero |NOCACHE. CACHE permite guardar en memoria un conjunto previamente asignado de números de secuencia para garantizar acceso más rápido. En secuencias cíclicas, este valor debe ser menor que el número de valores del ciclo. El mínimo valor es 2. NOCACHE indica que los valores de la secuencia no se pueden precalcular.

  Una vez creada la secuencia, accedemos a ella mediante las pseudocolumnas CURRVAL, que devuelve el valor actual de la secuencia, y NEXTVAL, que devuelve el siguiente valor e incrementa la secuencia. Para acceder a estos valores tenemos que poner el nombre de la secuencia, un punto y, a continuación, la pseudocolumna: NOMBRESECUENCIA.CURRVAL NOMBRESECUENCIA.NEXTVAL.



Caso práctico

3 Se crea una tabla llamada FRUTAS con dos columnas: CODIGO y NOMBRE. La columna CODIGO se define como clave primaria:

```
CREATE TABLE FRUTAS (CODIGO NUMBER(2) NOT NULL PRIMARY KEY, NOMBRE VARCHAR2(15));
```

Ahora se crea una secuencia llamada CODIGOS que generará números empezando por el valor 1, con incremento 1 y cuyo máximo valor para la secuencia será 99: **CREATE SEQUENCE** CODIGOS START WITH 1 INCREMENT BY 1 MAXVALUE 99;

Se insertan filas en la tabla FRUTAS usando la secuencia CODIGOS para generar el CODIGO de cada fila de la tabla:

```
INSERT INTO FRUTAS VALUES (CODIGOS.NEXTVAL, 'MANZANAS');
INSERT INTO FRUTAS VALUES (CODIGOS.NEXTVAL, 'NARANJAS');
INSERT INTO FRUTAS VALUES (CODIGOS.NEXTVAL, 'PERAS');
```

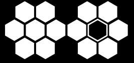
Para consultar el valor actual de la secuencia escribimos: SELECT **CODIGOS.CURRVAL** FROM DUAL;



Actividades propuestas

3 Crea una secuencia cíclica que comience en 1, se incremente en 1, y el máximo valor sea de 10. Usa la secuencia y comprueba que al llegar a 10 vuelve a iniciarse.

Para eliminar una secuencia de la base se usa la orden DROP SEQUENCE. Por ejemplo, borramos la secuencia creada anteriormente: **DROP SEQUENCE** CODIGOS;



B. Índices

Con los índices se acelera el tiempo de respuesta en las consultas. Un **índice** es un objeto de base de datos que se asocia a una tabla y al que se asocia una o varias columnas de la tabla. Se puede almacenar en un *tablespace* diferente al de la tabla que indexa. Contienen un elemento para cada valor que aparece en la columna o columnas indexadas de la tabla, y proporciona un acceso rápido y directo a las filas mediante el ROWID.

Estudiaremos la utilidad de los índices con un ejemplo: Se supone que Hacienda dispone de una tabla de CONTRIBUYENTES en la que cada fila se identifica por el NIF del contribuyente. Para consultar un NIF determinado realizamos la siguiente consulta: `SELECT * FROM CONTRIBUYENTES WHERE NIF='7866978-A';`

Si la columna NIF no está indexada, Oracle recorre la tabla de CONTRIBUYENTES secuencialmente, desde la primera fila hasta el final de la tabla, tanto si encuentra como si no encuentra el dato que se busca. Si la columna NIF está indexada, Oracle realizará una búsqueda binaria en el índice hasta encontrar el dato buscado, obteniendo el ROWID de la fila asociada mediante un único acceso sobre la tabla CONTRIBUYENTES.

Se debe indexar cuando se disponga de una gran cantidad de filas en una tabla. Cada fila se debe identificar con una o varias columnas. Es conveniente no indexar las tablas pequeñas, a no ser que sea necesario definir claves primarias o columnas únicas. No se deben indexar columnas que son modificadas (UPDATE) a menudo y que posean pocos valores diferentes. Tampoco es útil indexar tablas en las que se haga una gran cantidad de sentencias UPDATE, DELETE o INSERT, ya que la modificación de la tabla implica la modificación del índice.

El formato para crear un índice es el siguiente:

```
CREATE INDEX nombreindice
ON nombretabla (colum[ASC|DESC] [,colum[ASC|DESC]]...)
[STORAGE clausulas_almacenamiento]
[TABLESPACE nombretablespace]
[otras_cláusulas];
```

Donde:

- *Nombretabla* es la tabla que se va a indexar.
- *colum* es la columna o columnas que se indexan.
- *ASC|DESC* se especifica para definir índices ascendentes o descendentes.

No hay que olvidar que cuando se hace una restricción de clave primaria (PRIMARY KEY) o una restricción de unicidad (UNIQUE) se crea un índice con el nombre de la restricción. Por ejemplo, para crear un índice en la columna EMP_NO para la tabla EMPLE escribiremos: `CREATE INDEX INDICEEMPLE ON EMPLE (EMP_NO);`

Las vistas USER_INDEXES y DBA_INDEXES informan sobre los índices creados.

Para eliminar un índice de la base de datos se usa la orden DROP INDEX. Por ejemplo, borramos el índice creado anteriormente: `DROP INDEX INDICEEMPLE;`

C. Enlaces de bases de datos

Para explicar los enlaces de bases de datos, vamos a suponer que disponemos en nuestra aula de dos servidores, un Servidor Novell y un Servidor Windows 2003, y que en ambos tenemos instalada la base de datos Oracle. Todos los usuarios tienen cuenta para conectarse a las dos bases de datos. Si nos conectamos desde un puesto a la base de datos que está en el Servidor Windows 2003 y queremos utilizar los datos de las tablas que están en la base de datos instalada en el Servidor Novell, tendremos que crear un enlace de base de datos o DATABASE LINK. Un **enlace de base de datos** es un objeto que permite acceder a objetos de una base de datos remota. Define el enlace entre una base de datos local y un nombre de usuario en una base de datos remota. Se usa para realizar consultas en tablas de la base de datos remota.

Para crear un enlace de base de datos se emplea la orden de SQL, CREATE DATABASE LINK, cuyo formato es:

```
CREATE [PUBLIC]DATABASE LINK nombredeenlace
CONNECT TO usuario IDENTIFIED BY  clave
USING 'cadena_de_conexión';
```

Donde:

- `PUBLIC` crea un enlace de base de datos público.
- *`cadena_de_conexión`* es la cadena de conexión utilizada para conectar con la base de datos remota.
- `usuario/clave` es el nombre de usuario y la contraseña utilizados para conectase a la base de datos remota. El nombre de usuario tiene que existir en la base de datos remota.

En el acceso a tablas remotas es necesario añadir al nombre de la tabla el nombre del enlace de la siguiente manera: NOMBRETABLA@NOMBREDEENLACE.



Caso práctico

4 **Para conectar a la base de datos remota (por ejemplo, está almacenada en un Servidor Novell), cuya cadena de conexión es 'DAI' y el nombre de usuario y su clave es MAJESUS/RAMOS, se crea el siguiente enlace de base de datos:**

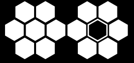
```
CREATE DATABASE LINK MIENLACE CONNECT TO MAJESUS IDENTIFIED BY RAMOS USING 'DAI';
```

Se puede acceder a la tabla EMPLE de MAJESUS de la siguiente manera: `SELECT * FROM EMPLE@MIENLACE;`

Para eliminar un enlace de base de datos se usa la orden DROP DATABASE LINK. Éste es su formato:

```
DROP [PUBLIC] DATABASE LINK nombredeenlace;
```

Por ejemplo, borramos el enlace anterior: `DROP DATABASE LINK MIENLACE;`



13.3 Copias de seguridad

Una de las funciones más importantes de un administrador de bases de datos es el mantener los datos siempre disponibles, proteger la base de datos de todos los posibles fallos y hacer que esté operativa lo antes posible ante cualquier fallo. Para ello, el DBA debe planificar y realizar copias de seguridad con regularidad. Las copias de seguridad son esenciales a la hora de recuperar la base de datos ante fallos inesperados.

Oracle ofrece una variedad de procedimientos de copia de seguridad y opciones que facilitan la protección de bases de datos. Puede haber *copias de seguridad lógicas* y *copias físicas*. Existen tres métodos: *exportaciones* o *copias lógicas*, *copias de seguridad fuera de línea* y *en línea*, que son *copias físicas*.

Por norma general, las bases de datos en producción basan su seguridad en copias físicas.

A. Copias de seguridad lógicas

Las **copias lógicas** consistirán en leer datos de la base de datos y extraerlos generando un archivo de exportación que podrá ser recuperado con la orden IMPORT de Oracle. Se pueden exportar usuarios, tablas y hasta la base de datos completa.

En general, las utilidades IMPORT y EXPORT de Oracle van a permitir:

- Archivar datos históricos. Por ejemplo, hacer copias de los esquemas cada vez que se realiza algún cambio.
- Guardar definiciones de tablas con o sin datos como medida de protección ante fallos de usuarios.
- Migrar datos de una versión a otra de Oracle cuando se actualiza una versión o mover datos entre máquinas con distinto sistema operativo y bases de datos diferentes.
- Transportar *tablespaces* de una base de datos a otra.

Utilidad EXPORT

Para ejecutar esta utilidad lo hacemos en modo comando desde el sistema operativo, sin entrar a SQL, PLUS. Para ver el formato y los parámetros del comando EXP escribimos:

```
C:\>exp help=yes
```

Los parámetros más comunes son:

- USERID: Nombre de usuario/contraseña de la cuenta que realiza la exportación. La palabra USERID es opcional.
- BUFFER: Tamaño del buffer de datos en bytes.
- FILE: Nombre del archivo de salida, por defecto es *expdat.dmp*.

- COMPRESS: Indica si la exportación debe comprimir segmentos fragmentados en una misma extensión. Se pone Y o N. Para tablas grandes se utiliza `compress=N`

- GRANTS: Especifica si las concesiones sobre los objetos se exportan (Y) o no (N). Por defecto es Y.

- INDEXES: Especifica si se exportan los índices de las tablas (Y) o no (N). Por defecto es Y.

- DIRECT: Indica si se realiza una exportación directa, sin utilizar la caché. Por defecto es N.

- LOG: Nombre del archivo para mensajes informativos y de error.

- ROWS: Especifica si se exportan las filas de las tablas (Y) o no (N). Por defecto es Y.

- CONSISTENT: Indica si se mantiene una versión con coherencia de lectura de todos los objetos exportados. Necesario cuando las tablas se actualizan mientras se están exportando.

- FULL: Se realiza una exportación completa de la base de datos si se pone `FULL=Y`. Por defecto es N.

Si se exportan datos de una instancia a otra hay que asegurarse de que la base de datos destino tenga los mismos nombres de espacios de tablas que la base de datos origen, pues los nombres de espacios de tablas se exportan como parte de la definición de los objetos.

- OWNER: Indica una lista de cuentas de la base de datos que se van a exportar.

- CONSTRAINTS: Especifica si se exportan las restricciones sobre tablas (Y) o no (N). Por defecto es Y.

- TRIGGERS: Especifica si se exportan los triggers (Y) o no (N). Por defecto es Y.

- TABLES: Especifica la lista de tablas que se va a exportar.

- PARFILE: Especifica el nombre del archivo de parámetros que se va a pasar a EXPORT. Puede contener entradas para todos estos parámetros.

- QUERY: Especifica una cláusula WHERE que se aplicará a cada tabla durante la exportación.

- TABLESPACES: Especifica los *tablespaces* que se deben transportar.

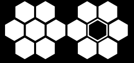
- TRANSPORT_TABLESPACE: Activa la exportación de *tablespaces* transportables.

- FEEDBACK: Visualiza el proceso de exportación cada X filas, por defecto es 0.

A continuación se muestran los modos de exportación de la orden EXPORT:

- **Modo tabla:** en este modo se exportan las definiciones de la o las tablas, los datos o las filas seleccionadas de la tabla, concesiones de tablas del propietario, índices de tablas del propietario y restricciones de tabla.

- **Modo usuario:** se exportan todos los objetos del esquema del usuario, es decir, definiciones de tablas, los datos de las tablas, procedimientos, funciones y paquetes. Concesiones del propietario, índices del propietario y restricciones de tablas.



– **Modo *tablespace:*** esta opción se puede utilizar para mover *tablespaces* de una base de datos a otra. El movimiento de datos mediante *tablespaces* transportables puede ser más rápido que hacer una importación y exportación de los datos puesto que para transportar un *tablespace* sólo es necesario copiar los archivos de datos e integrar su información estructural.

– **Modo base de datos completa:** este modo exporta todos los objetos de la base de datos excepto los del esquema SYS. No coge el diccionario de datos.

Utilidad IMPORT

Se usa para la recuperación de archivos de exportación. Se puede utilizar para recuperar objetos o usuarios seleccionados desde el archivo de volcado de exportación. Muchos de los parámetros del comando coinciden con EXPORT. Otros parámetros son:

– USERID: Nombre de usuario/contraseña de la cuenta que realiza la importación, la palabra *userid* es opcional, tienen que ser usuarios con privilegios DBA o IMP_FULL_DATABASE.

– FILE: Nombre del archivo de exportación a importar.

– SHOW: Especifica si el contenido del archivo debe mostrarse (S/N) en lugar de ejecutarse. Por defecto es N.

– IGNORE: Especifica si la importación debe ignorar errores encontrados (S/N) al ejecutar el comando CREATE. Se utiliza si los objetos a importar ya existen. Por defecto es N.

– FROMUSER: Lista de cuentas de la base de datos cuyos objetos deben leerse desde el archivo de exportación cuando `FULL=N`.

– TOUSER: Lista de cuentas de la base de datos en las que se importarán los objetos del archivo de exportación.

– DESTROY: Especifica si se ejecutan los comandos CREATE TABLESPACE que se encuentran en los archivos de exportaciones completas, destruyendo los archivos de datos contenidos en la base de datos en la que van a ser importados. Por defecto es N.

– INDEXFILE: Esta opción permite escribir todos los comandos CREATE TABLESPACE, CREATE CLUSTER y CREATE INDEX en un archivo en lugar de ejecutarlos. Si `CONSTRAINTS=Y`, entonces las restricciones se escribirán también en el archivo. Este archivo puede entonces ejecutarse tras importarlo con `INDEXES=N`, será necesario hacer pequeños cambios.

– RESUMABLE: Especifica si se continúa la sesión (S/N) después de los errores. Por defecto es N.

– COMPILE: Especifica si se deben compilar (S/N) de nuevo procedimientos, funciones y paquetes en la importación. Por defecto es N.

– DATAFILES: Indica la lista de archivos de datos que se va a transferir a la base de datos.

– TTS_OWNERS: Especifica el nombre o la lista de nombres de propietarios de datos en el espacio de tablas transportable.



Caso práctico

5 **Para este caso práctico es necesario ejecutar el script que se encuentra en el CD del libro y que crea los *tablespaces*: EJERCICIOS y TEMPEJER, los usuarios PRUEBA y USUEX_IM con privilegios para hacer EXPORT e IMPORT, y las tablas NOTAS, ALUMNOS, PROFESORES, ASIGNATURAS y ASIG_PROF, con sus relaciones y datos.**

- `C:\>EXP USERID=PRUEBA/PRUEBA`, crea el archivo *expdat.dmp* en la carpeta desde donde se ejecuta el comando, con el esquema completo del usuario. Para importar el archivo será necesario tener un usuario con privilegio IMP_FULL_DATABASE, o DBA.

  `IMP USUEX_IM/USUEX_IM FROMUSER=PRUEBA FILE C:\EXPDAT.DMP`, importa todo lo de prueba al esquema de USUEX_IM.

- `C:\>EXP USERID=PRUEBA/PRUEBA FILE=C:\EXPORT_IMPORT\3_SECUN.DMP TABLES =(ASIGNATURAS) QUERY = \"WHERE CURSO = 3 AND ETAPA= 'SECUNDARIA'\"`, crea el archivo 3_SECUN.DMP en la carpeta indicada, y contiene la tabla asignaturas con las asignaturas del curso tercero y de la etapa secundaria.

  `IMP USUEX_IM/USUEX_IM FROMUSER=PRUEBA FILE=C:\EXPORT_IMPORT\3_SECUN.DMP`, importa la tabla ASIGNATURAS de tercero de secundaria del usuario PRUEBA al usuario USUEX_IM.

- `C:\>EXP SCOTT/TIGER FILE = C:\EXPORT_IMPORT\DATOS_SCOT.DMP OWNER=SCOTT GRANTS=N INDEXES=Y COMPRESS=Y ROWS=Y, SCOTT` exporta todos sus objetos y crea el archivo DATOS_SCOT.DMP, no se importan concesiones sobre objetos. Se utiliza compresión.

  `IMP USUEX_IM/USUEX_IM FROMUSER = SCOTT TOUSER=PRUEBA INDEXES=Y ROWS=Y FILE= C:\EXPORT_IMPORT\DATOS_SCOT.DMP`. El usuario USUEX_IM importa todo lo del usuario SCOTT al usuario PRUEBA, incluidos índices y filas.

  `C:\>IMP USUEX_IM/USUEX_IM FULL=Y NDEXFILE=C:\EXPORT_IMPORT\INDEXSCOT.SQL FILE=C:\EXPORT_IMPORT\DATOS_SCOT.DMP`. En este caso, se crea el archivo de índice INDEXSCOT.SQL, a partir del archivo de exportación. Este archivo se puede ejecutar aparte desde SQL haciendo algunos cambios en el mismo. Si se hace la importación sin índices pondremos: `IMP USUEX_IM/USUEX_IM FROMUSER = SCOTT TOUSER=PRUEBA INDEXES=N ROWS=Y FILE=C:\EXPORT_IMPORT\DATOS_SCOT.DMP`.

- `C:\>EXP TRANSPORT_TABLESPACE=Y TABLESPACES = (EJERCICIOS) FILE = C:\EXPORT_IMPORT\TABL_EJERCI.DMP LOG=C:\EXPORT_IMPORT\LOGTAB_EJER.LOG.`

  Exporta el *tablespace* EJERCICIOS. Este EXPORT debe hacerlo SYS o SYSTEM y conectarse como SYSDBA. Para poderlo transportar debe estar en modo READ ONLY. Es decir, antes el SYS debe ponerlo en sólo lectura:

  ```
  SQL> CONNECT SYS/ARM AS SYSDBA
  SQL> ALTER TABLESPACE EJERCICIOS READ ONLY;
  ```

  Una vez realizada la exportación poner el *tablespace* en modo lectura escritura. `SQL>ALTER TABLESPACE EJERCICIOS READ WRITE;`

- `C:\>EXP FULL=Y FILE=C:\BDCOMPLETA.DMP`. Se hace una copia completa de la base de datos. Cuando pida usuario nos conectamos con SYS como SYSDBA:



B. Copias de seguridad físicas

Éstas consisten en copiar los archivos que constituyen la base de datos. También se las denomina *copias de seguridad del sistema de archivos* ya que implican la utilización de comandos de copia de seguridad del sistema operativo. Oracle soporta dos métodos de copia de seguridad física: *fuera de línea* o *en frío*, y *en línea* o *en caliente*.

Las **copias de seguridad fuera de línea o en frío** se producen cuando la base de datos se ha desconectado de modo normal y no debido a ningún fallo, es decir, la base de datos se ha cerrado con uno de estos comandos SHUTDOWN, SHUTDOWN IMMEDIATE o SHUTDOWN TRANSACTIONAL. Los archivos que deben copiarse son:

- Todos los archivos de datos.
- Todos los archivos de control.
- Todos los archivos de Redo Logs.

Y opcionalmente podemos copiar los archivos de parámetros de inicialización de la base de datos init.ora y spfile.ora. Hacer una copia de todos estos archivos proporciona una imagen exacta de la base de datos.

Así pues, para hacer una copia de una base de datos, primero hay que hacer SHUTDOWN y luego copiar en el destino elegido los archivos de datos que normalmente suelen ubicarse en: `ORACLE\PRODUCT\10.1.0\ORADATA\ORCL`.

Y los de parámetros en: `ORACLE\PRODUCT\10.1.0\ADMIN\ORCL\PFILE`, los pfile, y en `ORACLE\PRODUCT\10.1.0\DB_1\DATABASE`, los spfile.

Para saber los archivos de datos a copiar y su ubicación los consultamos en las siguientes vistas:

```
SQL> select name from v$datafile;
NAME
-------------------------------------------------------
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\SYSTEM01.DBF
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\UNDOTBS01.DBF
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\SYSAUX01.DBF
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\EXAMPLE01.DBF
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\EJERCIC.DBF

SQL> select name from v$controlfile;
NAME
-------------------------------------------------------
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\CONTROL01.CTL
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\CONTROL02.CTL
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\CONTROL03.CTL
```

```
SQL> select  MEMBER  from v$logfile;
MEMBER
-------------------------------------------------------
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\REDO03.LOG
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\REDO02.LOG
C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\REDO01.LOG
```

Para restaurar esta base de datos copiamos la copia de seguridad realizada en los directorios correspondientes e iniciamos la base de datos con el comando STARTUP. La base de datos estará cerrada.

Las **copias de seguridad en línea** o **en caliente** se realizan con la base de datos abierta, para ello la base de datos debe estar en modo ARCHIVELOG para que los Redo Log se archiven creando registro de todas las transacciones realizadas. Recuerda que Oracle escribe en los Redo Log de manera cíclica: cuando se llena uno, pasa al siguiente, y así sucesivamente. Antes de sobrescribir cualquiera de ellos, el proceso ARCH realiza una copia de ese Redo Log. La mayoría de las bases de datos en producción deben ejecutarse en modo ARCHIVELOG. Mientras la base de datos está abierta puede realizarse una copia de:

- Todos los archivos de datos.
- Todos los archivos de Redo Logs archivados.
- Un archivo de control, mediante el comando **ALTER DATABASE**.

Para utilizar la función ARCHIVELOG, hay que poner la base de datos en modo ARCHIVELOG. Antes comprobamos el modo con `archive log list` (para todo esto hay que conectarse con SYS modo SYSDBA):

```
SQL> archive log list
Database log mode              No Archive Mode
Automatic archival             Disabled
Archive destination            USE_DB_RECOVERY_FILE_DEST
Oldest online log sequence      226
Current log sequence           228
```

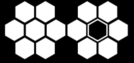
Por defecto, la base de datos está en modo NOARCHIVELOG, es decir, con la recuperación de medios deshabilitada. Para ponerla en ARCHIVELOG la base de datos debe estar cerrada y montada. Los pasos para pasar a modo ARCHIVELOG son:

1.º Se cierra la base de datos:

```
SQL> shutdown
Database closed.
Database dismounted.
ORACLE instance shut down.
```

2.º Se monta la base de datos por defecto, la instancia ORCL. El *init* está en el directorio `C:\ORACLE\PRODUCT\10.1.0\ADMIN\ORCL\PFILE`:

```
SQL> startup mount;
```



```
ORACLE instance started.

Total System Global Area  171966464 bytes
Fixed Size                   787988 bytes
Variable Size             145750508 bytes
Database Buffers           25165824 bytes
Redo Buffers                 262144 bytes
Database mounted.
```

Comprobamos que se monta la instancia ORCL:

```
SQL> select name from v$database;

NAME
---------
ORCL
```

3.º Activamos el modo ARCHIVELOG:

```
SQL> ALTER DATABASE ARCHIVELOG;
Database altered.
```

4.º A continuación, abrimos la base de datos y comprobamos si ha cambiado su estado:

```
SQL> ALTER DATABASE OPEN;
Database altered.

SQL> ARCHIVE LOG LIST;

Database log mode              Archive Mode
Automatic archival             Enabled
Archive destination            USE_DB_RECOVERY_FILE_DEST
Oldest online log sequence     229
Next log sequence to archive   231
Current log sequence           231
```

La base de datos permanecerá en este modo hasta que se ponga en NOARCHIVELOG. La ubicación de los archivos de Redo Log archivados se determinan por el parámetro DB_RECOVERY_FILE_DEST. Si editamos el *pfile* vemos que su ubicación es: `C:\oracle\product\10.1.0\flash_recovery_area`.

Para comprobar que se realizan copias de los archivos de Redo Log, hacemos cambios de log de forma manual:

```
SQL> ALTER SYSTEM SWITCH LOGFILE;
SYSTEM ALTERED.
```

Luego comprobamos en el directorio `C:\oracle\product\10.1.0\flash_recovery_area` lo que se ha creado. Observa que se ha creado una carpeta con el nombre de la instancia, con una subcarpeta para ARCHIVELOG. En la Figura 13.1 se muestran varios Redo Logs archivados.

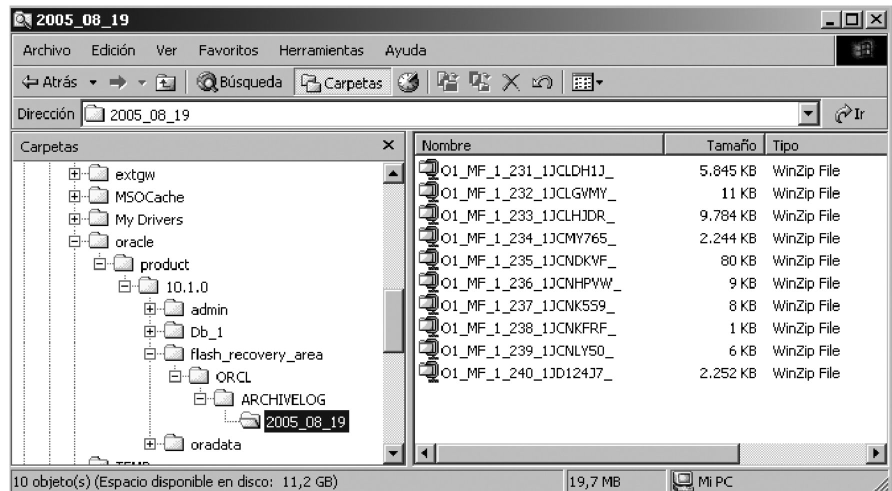


Figura 13.1. *Ubicación de los Redo Logs archivados.*

Para identificar a los archivos Oracle pone la extensión .ARC y un nombre, seguido del número de secuencia coincidente con `Current log sequence`. Se enumeran en el orden en que se crean.

Si hacemos un ARCHIVE LOG LIST, observamos que ha cambiado la secuencia y el número de secuencia actual, con respecto a la anterior ejecución. La vista V$LOG indicará el log actual, y dirá si se han archivado a no:

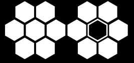
```
SQL> SELECT * FROM V$LOG;

GROUP#  THREAD#  SEQUENCE#     BYTES  MEMBERS  ARC    STATUS  FIRST_CHANGE#  FIRST_TI
     1        1        233  10485760        1  YES  INACTIVE        5215596  19/08/05
     2        1        234  10485760        1   NO   CURRENT        5215902  19/08/05
     3        1        232  10485760        1  YES  INACTIVE        5215561  19/08/05

SQL> ARCHIVE LOG LIST;
Database log mode              Archive Mode
Automatic archival             Enabled
Archive destination            USE_DB_RECOVERY_FILE_DEST
Oldest online log sequence     232
Next log sequence to archive   234
Current log sequence           234
```

Una vez que la base de datos está en modo ARCHIVELOG y se ha provocado un cambio de log, se procederá a realizar las copias de seguridad de los *tablespaces*, de los archivos de Redo Log y del archivo de control.

- **Proceso para la copias de los *tablespaces***
  - Poner el *tablespace* en estado de copia ALTER TABLESPACE nombre BEGIN BACKUP. Esta orden hace que se congele la cabecera del archivo de datos y que se mantenga el número de secuencia, con lo que en caso de recuperación los logs se aplican a partir de este número.



Aunque esté en modo BACKUP, el archivo de datos sigue estando disponible para el funcionamiento normal.

– Hacer copia de los archivos de datos asociados al *tablespace* (copiar y pegar).

– Dejar el *tablespace* en modo normal: ALTER TABLESPACE nombre END BACKUP.

– Forzar un punto de control para sincronizar las cabeceras de archivos mediante un cambio de log: ALTER SYSTEM SWITCH LOGIFLE.

Todos estos pasos se repiten para cada *tablespace*.

Caso práctico

6 Realización de la copia de seguridad del *tablespace* USERS. Antes crea un directorio para los backups en `C:\BACKUPS`.

– Antes de hacer la copia vemos el fichero de datos asociado para luego copiarlo.

```
SQL> SELECT FILE_ID,FILE_NAME,TABLESPACE_NAME FROM DBA_DATA_FILES;

FILE_ID     FILE_NAME                                                TABLESPACE_NAME
-------     ------------------------------------------------         --------
      4     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF                  USERS
      3     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\SYSAUX01.DBF                SYSAUX
      2     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\UNDOTBS01.DBF             UNDOTBS1
      1     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\SYSTEM01.DBF                SYSTEM
      5     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\EXAMPLE01.DBF              EXAMPLE
      6     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\EJERCIC.DBF             EJERCICIOS
```

– Iniciamos el backup con la orden:

```
SQL> ALTER TABLESPACE USERS BEGIN BACKUP;
Tablespace altered.
```

– Copiamos el fichero `C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF` en el directorio creado. La vista dinámica, con información de la copia de seguridad V$BACKUP, nos devuelve los archivos que están en modo BACKUP, cuyo estado cambia a modo ACTIVE. La columna CHANGE# indica el número de *checkpoint*:

```
SQL> SELECT * FROM V$BACKUP;

     FILE#      STATUS                   CHANGE#       TIME
----------      ------------------    ----------       --------
         1      NOT ACTIVE                     0
         2      NOT ACTIVE                     0
         3      NOT ACTIVE                     0
         4      ACTIVE                   5218939       19/08/05
         5      NOT ACTIVE                     0
         6      NOT ACTIVE                     0
```

(Continúa)

(Continuación)

– Terminada la copia, ponemos el *tablespace* a estado normal. La vista V$BACKUP registrará los cambios.

```
SQL>  ALTER TABLESPACE USERS END BACKUP;
Tablespace altered.
```

- **Proceso para las copias de los Redo Logs archivados**
  Hacer copia de los Redo Log archivados ubicados en el directorio de archivelog. Después de la copia se pueden borrar.

- **Proceso para las copias del archivo de control**
  Este archivo es uno de los que más debe protegerse. Su información es necesaria para el inicio de una instancia. La información de este archivo, como el archivo de Redo Log online actual o los nombres de los archivos de bases de datos, es necesaria para la recuperación de la instancia. Es conveniente mantener una copia de este archivo cada vez que hay un cambio en la configuración de los ficheros de datos. Los pasos para su copia son:

  1.º Se crea una imagen del archivo con ALTER DATABASE BACKUP CONTROLFILE TO <*archivo_destino*>

  2.º Se debe crear un archivo de comandos para reproducir el archivo de control con ALTER DATABASE BACKUP CONTROLFILE TO TRACE. Este archivo de traza permite recrear el archivo de control y se crea en el directorio indicado por el parámetro USER_DUMP_DEST.



Caso práctico

7 Realiza el backup del archivo de control.

– Primero creamos el archivo de traza, pero antes comprobamos la ubicación visualizando el parámetro USER_DUMP_DEST:

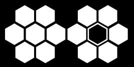
```
SQL> SHOW PARAMETER USER_DUMP_DEST
NAME                  TYPE        VALUE
-----------------     -------     ----------------------------------------
user_dump_dest        string      C:\ORACLE\PRODUCT\10.1.0\ADMIN\ORCL\UDUMP

SQL> ALTER DATABASE BACKUP CONTROLFILE TO TRACE;
Database altered.
```

Se crea el fichero de traza. Edita el archivo para ver su contenido. Su nombre suele llevar el nombre de la instancia, seguido de `ora` y un número, por ejemplo: orcl_ora_876.

– En segundo lugar, se hace la copia que guardaremos en el directorio `C:\BACKUPS`:

```
SQL>  ALTER DATABASE BACKUP CONTROLFILE TO 'C:\BACKUPS\CCONTROL.BAK';
Database altered.
```



13.4 Restauración de datos

A la hora de recuperar los datos, ante fallos imprevistos, hay que tener en cuenta el modo en el que estaba la base de datos y las copias de que disponemos. El modo NOARCHIVELOG puede ser adecuado en bases de datos dedicadas a desarrollo, entrenamiento y prueba. En este caso, la restauración es sencilla. Basta con restaurar los archivos de la copia de seguridad en la ubicación de los archivos de la base de datos. El tiempo de recuperación es el tiempo que el hardware tarde en la restauración. El inconveniente de este modo es que los datos introducidos desde la última copia se pierden y sería necesario aplicarlos de nuevo.

Caso práctico



8 **En este caso práctico, se va a simular un fallo en un archivo de datos: al abrir la base de datos ocurrirá un error. Para recuperarla procederemos a restaurar el archivo correspondiente. Antes de nada, comprueba que la base de datos está en NOARCHIVELOG; si no es así ponerla en NOARCHIVELOG.**

– En primer lugar, cerrar la base de datos con SHUTDOWN IMMEDIATE. Una vez cerrada hacemos copia de los archivos de datos (ya vimos cómo hacerlo en el apartado «Copias de seguridad en frío»).

– Seguidamente vamos a simular un fallo en el archivo USERS01.DBF. Por ejemplo, lo cambiamos de nombre o lo borramos.

– Arrancamos la base de datos con el comando STARTUP y ocurre lo siguiente:

```
SQL> startup
ORACLE instance started.

Total System Global Area  171966464 bytes
Fixed Size                   787988 bytes
Variable Size             145750508 bytes
Database Buffers           25165824 bytes
Redo Buffers                 262144 bytes
Database mounted.
ORA-01157: cannot identify/lock data file 4 - see DBWR trace file
ORA-01110: data file 4: 'C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF'
```

La base de datos llega a montarse, pero no se abre porque no encuentra un fichero de datos.

– Lo siguiente sería restaurar el archivo USERS01.DBF de los archivos de copia de seguridad. Basta con copiar y pegar.

– Cerrar la base de datos SHUTDOWN IMMEDIATE y volverla a abrir STARTUP. La base de datos abrirá sin ningún problema.

Nos imaginamos ahora que la restauración de este archivo se hace en una ubicación diferente porque el disco se ha estropeado. Lo hacemos en la carpeta `D:\BASEORCL`. En este caso, primero, se monta la instancia, luego utilizamos el comando ALTER DATABASE RENAME FILE para indicar la nueva ubicación de USERS01.DBF. De esta forma, el archivo de control se actualiza. Seguidamente se abre la base de datos:

```
SQL> STARTUP MOUNT
SQL> ALTER DATABASE RENAME FILE
```

(Continúa)

(Continuación)

```
'C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF'
TO 'D:\BASEORCL\USERS01.DBF';
SQL> ALTER DATABASE OPEN;
```

Ojo: al hacer esto hemos cambiado la base de datos. El fichero USERS01.DBF está en su nueva ubicación. Si se desea colocarlo en la ubicación anterior hay que repetir estos pasos para ubicarlo en la carpeta habitual: parar la base de datos, hacer la copia en frío de `D:\BASEORCL\USERS01.DBF` y copiarlo en su carpeta habitual, montar la base de datos y hacer un `ALTER DATABASE RENAME FILE 'D:\BASEORCL\USERS01.DBF' to 'C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF'`. Si esto no se hace así ocurrirá un error de sincronización y Oracle pedirá recuperación de datos:

```
SQL> ALTER DATABASE OPEN;
ALTER DATABASE OPEN
*
ERROR at line 1:
ORA-01113: file 4 needs media recovery
ORA-01110: data file 4:
'C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF'
SQL> recover datafile
'C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF';
Media recovery complete.
SQL> ALTER DATABASE OPEN;
Database altered.
```

Oracle recuperará todas las transacciones que se encuentren en los archivos de redo (REDO01.LOG, REDO02.LOG, REDO03.LOG), sin embargo, si estos archivos se sobrescriben y la base de datos está en NOARCHIVELOG se perderán las transacciones no registradas en ellos.

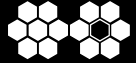
A. Recuperación en modo ARCHIVELOG

En el modo ARCHIVELOG podemos hacer recuperaciones completas de la base de datos y recuperaciones hasta un punto determinado.

En las recuperaciones completas es necesario tener todos los archivos de Redo Logs archivados desde que se realizó la copia de seguridad que se está utilizando en la restauración. Si falta alguno no se podrá realizar una restauración completa pues se aplican todos los archivos en secuencia. Este tipo de copia tiene las siguientes ventajas:

- Sólo es necesario restaurar los archivos perdidos o corruptos.
- Recupera todos los datos hasta el momento del fallo.
- El tiempo de recuperación es el que se tarda en restaurar los archivos perdidos y en aplicar todos los archivos de log archivados.

- **Recuperación completa ante fallos del medio físico**
  Antes de hacer recuperaciones de datos, en caso de errores, corrupción de discos y archivos de datos, es necesario disponer de una copia válida de los archivos de datos, los

logs desde que se realizó la copia de seguridad y los archivos de Redo Logs que aún no están archivados. Además, la base de datos debe estar definida en modo ARCHIVELOG.

Cumpliendo estos requisitos, el proceso para recuperar los archivos de datos es el siguiente:

- Comprueba que los archivos que se van a sobrescribir no están abiertos durante la recuperación. Consulta las vistas: V$DATAFILE y V$TABLESPACE para comprobar el estado del archivo.
- Sólo se debe restaurar el archivo perdido o dañado.
- Los archivos de Redo Log online no se restauran.
- La base de datos se pondrá en modo MOUNT o en modo OPEN.
- Utilizaremos el comando RECOVER para recuperar los datos:
    - Con la base de datos montada podremos utilizar: RECOVER DATABASE, RECOVER DATAFILE o ALTER DATABASE RECOVER DATABASE.
    - Con la base de datos abierta podremos utilizar: RECOVER TABLESPACE, RECOVER DATAFILE o ALTER DATABASE RECOVER DATAFILE.

Para realizar una recuperación completa utilizaremos los siguientes métodos:

- **Recuperación con la base de datos abierta.** En este caso, se pierde un archivo de datos al que no pertenecen ni el sistema ni el *tablespace* de Rollback. Los ficheros de datos se pondrán OFFLINE, arrancará la base de datos, se realizará la recuperación de ese archivo, y seguidamente el archivo se pondrá ONLINE.

Caso práctico



9 Recuperación de un *datafile* con la base de datos abierta. Antes de realizar el caso práctico vamos a comprobar el modo de apertura de la base de datos con ARCHIVE LOG LIST o consultando V$DATABASE. También hay que tener un backup de todos los *tablespaces*, como se hizo en el apartado de copias en caliente, con la base de datos arrancada.

- En primer lugar, se hacen copias de los archivos de datos y los *tablespaces* con el usuario SYS, para ello consultamos la vista: `SQL> SELECT FILE_ID,FILE_NAME,TABLESPACE_NAME FROM DBA_DATA_FILES`.
- Iniciamos el backup de los *tablespaces:*

```
SQL> ALTER TABLESPACE USERS BEGIN BACKUP;
SQL> ALTER TABLESPACE SYSAUX BEGIN BACKUP;
SQL> ALTER TABLESPACE UNDOTBS1 BEGIN BACKUP;
SQL> ALTER TABLESPACE SYSTEM BEGIN BACKUP;
SQL> ALTER TABLESPACE EXAMPLE BEGIN BACKUP;
```

- Copiamos los ficheros asociados a los *tablespaces* en el directorio de copias `C:\BACKUPS`.

(Continúa)

(Continuación)

– Una vez copiados, finalizamos la copia poniendo:

```
SQL> ALTER TABLESPACE USERS END BACKUP;
SQL> ALTER TABLESPACE SYSAUX END BACKUP;
SQL> ALTER TABLESPACE UNDOTBS1 END BACKUP;
SQL> ALTER TABLESPACE SYSTEM END BACKUP;
SQL> ALTER TABLESPACE EXAMPLE END BACKUP;
```

– Creamos la tabla NUEEMPLE en el esquema de Scott, a partir de la tabla EMP. Vamos a ir añadiendo registros y provocando cambios de log. Iremos apuntando los números de secuencia de los mismos para cuando se desea recuperar a partir de un log.

```
CREATE TABLE SCOTT.NUEEMPLE AS SELECT * FROM SCOTT.EMP;
```

Comprobamos el *tablespace* donde se ha creado esta tabla para asegurarnos que es USERS y para saber el fichero de datos a recuperar:

```
SQL> SELECT TABLESPACE_NAME FROM DBA_TABLES WHERE TABLE_NAME = 'NUEEMPLE' AND OWNER
= 'SCOTT';

SQL> INSERT INTO SCOTT.NUEEMPLE SELECT * FROM SCOTT.EMP;
SQL> COMMIT;
SQL> SELECT COUNT(*) FROM SCOTT.NUEEMPLE;
   COUNT(*)
 ----------
         28
SQL> SELECT * FROM V$LOG;
GROUP#  THREAD# SEQUENCE#     BYTES MEMBERS  ARC   STATUS  FIRST_CHANGE#  FIRST_TI
------  ------- ---------     -----    ----  ---   ------  -------------  --------
     1        1       251  10485760       1   NO   ACTIVE        5331863  22/08/05
     2        1       249  10485760       1  YES INACTIVE        5302660  22/08/05
     3        1       250  10485760       1   NO  CURRENT        5324174  22/08/05
```

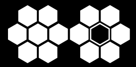
En la secuencia de log 250, el CURRENT, SCOTT.NUEEMPLE tiene 28 registros. Hacemos COMMIT, cambiamos de log y los apuntamos para luego recuperar la tabla en un momento determinado.

```
SQL> COMMIT;
SQL> ALTER SYSTEM SWITCH LOGFILE;
SQL> INSERT INTO SCOTT.NUEEMPLE SELECT * FROM SCOTT.EMP;
SQL> SELECT COUNT(*) FROM SCOTT.NUEEMPLE;
SQL> SELECT * FROM V$LOG;
SQL> COMMIT;
```

En la secuencia 251 la tabla SCOTT.NUEEMPLE tiene 42 registros.

```
SQL> ALTER SYSTEM SWITCH LOGFILE;
SQL> INSERT INTO SCOTT.NUEEMPLE SELECT * FROM SCOTT.EMP;
SQL> SELECT COUNT(*) FROM SCOTT.NUEEMPLE;
SQL> SELECT * FROM V$LOG;
SQL> COMMIT;
```

(Continúa)



(Continuación)

En la secuencia 252 la tabla SCOTT.NUEEMPLE tiene 56 registros. Si se abre la carpeta `C:\ORACLE\PRODUCT\10.1.0\FLASH_RECOVERY_AREA\ORCL\ARCHIVELOG\fecha_archivado\` se podrán ver los logs creados. En la Figura 13.2 se ven los logs del ejercicio. La fecha es la del 22 de agosto de 2005:

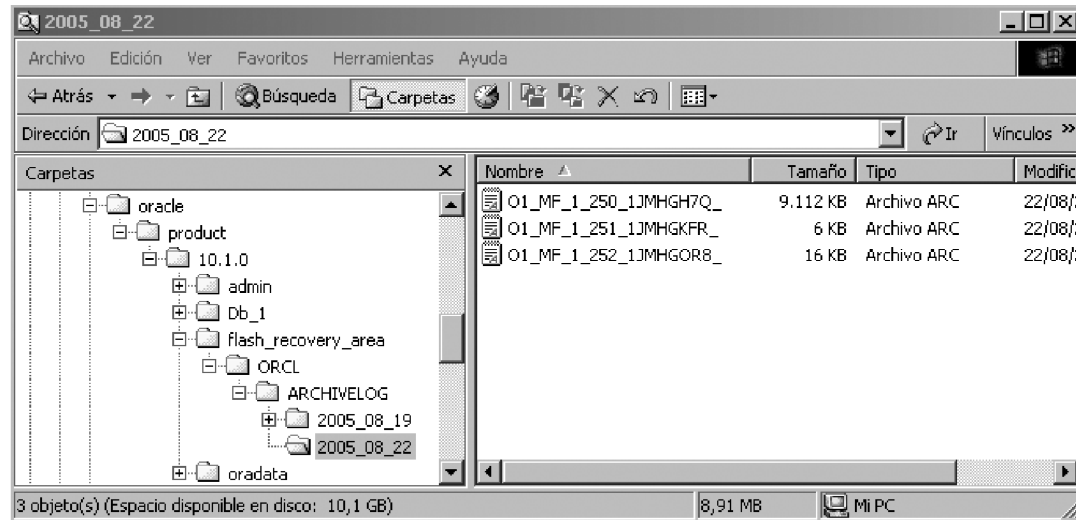


Figura 13.2. *Logs archivados del ejercicio.*

– A continuación, cerramos la base de datos con SHUTDOWN IMMEDIATE y provocamos un error en la base de datos: borramos el achivo USERS01.DBF, arrancamos y vemos que la base de datos se monta pero no arranca pues no localiza el archivo. Lo que hay que hacer es poner el *datafile* fuera de línea OFFLINE, restaurar la copia de seguridad, abrir la base de datos y recuperar el archivo:

```
SQL> ALTER DATABASE DATAFILE
'C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF' OFFLINE;
```

O también **ALTER DATABASE DATAFILE 4 OFFLINE;** (pues 4 es el número de este archivo). Al poner el archivo OFFLINE, permite que los usuarios que no trabajen con este archivo puedan conectarse y realizar transacciones.

Restauramos la copia de seguridad y abrimos la base de datos.

```
SQL> ALTER DATABASE OPEN;
```

Ponemos el archivo a ONLINE, pero ocurre un error y Oracle indica que el archivo se tiene que recuperar:

```
SQL> ALTER DATABASE DATAFILE 4 ONLINE;
ALTER DATABASE DATAFILE 4 ONLINE
*
ERROR at line 1:
ORA-01113: file 4 needs media recovery
ORA-01110: data file 4: 'C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF'
```

Se restaura con la orden RECOVER DATAFILE, para ver los archivos de datos que necesitan recuperación consultamos la vista: SELECT * FROM V$RECOVER_FILE;

(Continúa)

(Continuación)

```
SQL> RECOVER DATAFILE 4
ORA-00279: change 5331190 generated at 08/22/2005 13:42:39 needed for thread 1
ORA-00289: suggestion :
C:\ORACLE\PRODUCT\10.1.0\FLASH_RECOVERY_AREA\ORCL\ARCHIVELOG\2005_08_22\O1_MF_1_2
50_%U_.ARC
ORA-00280: change 5331190 for thread 1 is in sequence #250
Specify log: {<RET>=suggested | filename | AUTO | CANCEL}
Log applied.
Media recovery complete.
```

Aparece un mensaje en el que Oracle nos pregunta a partir de qué log va a aplicar para iniciar la recuperación. En el ejemplo, Oracle sugiere que empecemos aplicar el log 250 (es el primer log que se creó y archivó anteriormente), pulsamos *Enter* para aceptarlo, y esperamos a que se realice la restauración completa.

– A continuación, se pone ONLINE el *datafile* y comprobamos el contenido de la tabla NUUEMPLE de Scott, debe tener 56 registros.

```
SQL> ALTER DATABASE DATAFILE 4 ONLINE;
SQL> SELECT COUNT(*) FROM SCOTT.NUEEMPLE;
```

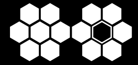
La tabla NUEEMPLE contiene todos los registros, ya que hemos aplicado todos los log.

– Podemos recuperar a partir de un log determinado. Sin embargo, al estar la base en modo ARCHIVELOG va a pedir toda la secuencia de log. En el ejemplo que tenemos, si deseamos recuperar a partir del log 251, nos dará error y nos indicará que necesita el 250 para recuperar el 251. En `Specify log` indicamos la ruta y el nombre de log:

```
SQL> recover datafile 4
ORA-00279: change 5331190 generated at 08/22/2005 13:42:39 needed for thread 1
ORA-00289: suggestion :
C:\ORACLE\PRODUCT\10.1.0\FLASH_RECOVERY_AREA\ORCL\ARCHIVELOG\2005_08_22\O1_MF_1_2
50_%U_.ARC
ORA-00280: change 5331190 for thread 1 is in sequence #250
Specify log: {<RET>=suggested | filename | AUTO | CANCEL}
C:\ORACLE\PRODUCT\10.1.0\FLASH_RECOVERY_AREA\ORCL\ARCHIVELOG\2005_08_22\O1_MF_1_2
51_1JMHGKFR_.ARC
ORA-00310: archived log contains sequence 251; sequence 250 required
ORA-00334: archived log:
'C:\ORACLE\PRODUCT\10.1.0\FLASH_RECOVERY_AREA\ORCL\ARCHIVELOG\2005_08_22\O1_MF_1_
251_1JMHGKFR_.ARC'
```

Recuperar el archivo y ponerlo ONLINE.

– **Recuperación de una base de datos cerrada**. Este método utiliza los comandos RECOVER DATABASE o RECOVER DATAFILE cuando la base de datos puede dejar de funcionar durante un tiempo, o cuando los archivos recuperados pertenecen al sistema o al *tablespace* del segmento de Rollback, o cuando es necesario recuperar toda la base de datos o la mayoría de los archivos de datos.



13.5 Copias de seguridad y recuperación con RMAN

RMAN es una utilidad que permite hacer rápidamente backup de todos los datos o de una parte de la base de datos. Es un potente lenguaje de comandos independiente del sistema operativo. Utiliza los procesos de Oracle para realizar las copias y las recuperaciones, de ahí que se denominen *operaciones de copia de seguridad y recuperación gestionadas por el servidor*. El RMAN tiene 4 componentes:

- **El ejecutable RMAN,** que se encuentra en el directorio bin de la carpeta de Oracle, y que se ejecuta desde el sistema operativo simplemente tecleando `rman`. Se pueden ver los parámetros y argumentos tecleando RMAN HELP. Ver Figura 13.3.

```
Seleccionar C:\WINNT\system32\cmd.exe

C:\>rman help

Argument     Value          Description
-----------------------------------------------------------------------------
target       quoted-string  connect-string for target database
catalog      quoted-string  connect-string for recovery catalog
nocatalog    none           if specified, then no recovery catalog
cmdfile      quoted-string  name of input command file
log          quoted-string  name of output message log file
trace        quoted-string  name of output debugging message log file
append       none           if specified, log is opened in append mode
debug        optional-args  activate debugging
msgno        none           show RMAN-nnnn prefix for all messages
send         quoted-string  send a command to the media manager
pipe         string         building block for pipe names
timeout      integer        number of seconds to wait for pipe input
-----------------------------------------------------------------------------
Both single and double quotes (' or ") are accepted for a quoted-string.
Quotes are not required unless the string contains embedded white-space.

RMAN-00571: ===========================================================
RMAN-00569: =============== ERROR MESSAGE STACK FOLLOWS ===============
RMAN-00571: ===========================================================
RMAN-00552: syntax error in command line arguments
RMAN-01005: syntax error: found "identifier": expecting one of: "append, at, aux
iliary, catalog, cmdfile, clone, debug, log, msglog, mask, msgno, nocatalog, pip
e, rcvcat, script, slaxdebug, send, target, timeout, trace"
RMAN-01008: the bad identifier was: help
RMAN-01007: at line 2 column 1 file: command line arguments

C:\>
```

Figura 13.3. *Argumentos de RMAN.*

- **Una o más bases de datos de destino,** que son las bases de datos para las que se realizan operaciones de copia y restauración. La información del archivo de control de la base de datos de destino es utilizada por los procesos de servidor que llama RMAN en las operaciones de copia y restauración. La base de datos se especifica con el argumento TARGET. Pondremos RMAN TARGET ORCL para conectarnos a la instancia ORCL. El usuario que se conecta debe tener privilegios SYSDBA, si no ponemos usuario presupone SYS:

```
C:\>RMAN TARGET ORCL
Recovery Manager: Release 10.1.0.2.0 - Production
Copyright (c) 1995, 2004, Oracle.  All rights reserved.
target database Password:
connected to target database: ORCL (DBID=1086216171)
```

  O también: **`C:\>RMAN TARGET SYS/ARM@ORCL`**

- **El catálogo de recuperación de RMAN** también llamado **Metadatos de RMAN.** Los datos que RMAN utiliza para las operaciones de copia, recuperación y restauración, conocidos como los metadatos de RMAN, se almacenan en el archivo de control de la base de datos destino. Por esta razón, no es necesario crear un catálogo de recuperación. Sin embargo, resulta útil configurarlo pues muchas de las funciones, como los archivos de comandos almacenados y la copia de seguridad automática, no están disponibles sin el catálogo de recuperación.

  Se recomienda ubicarlo en una base de datos distinta a la de destino; en este caso, se crea en la misma base de datos. El catálogo lo debe crear un usuario con el rol RECOVERY_CATALOG_OWNER, el administrador creará el usuario RMAN, por ejemplo:

```
SQL> CREATE USER RMAN IDENTIFIED BY RMAN
TEMPORARY TABLESPACE TEMP
DEFAULT TABLESPACE USERS;
SQL> GRANT RECOVERY_CATALOG_OWNER, CONNECT, RESOURCE
TO RMAN;
SQL> ALTER USER RMAN QUOTA UNLIMITED ON USERS;
```

  Para crear el catálogo entramos como usuario RMAN a la instancia ORCL. Se crea un archivo de log CATALOG.LOG para ver si la creación se ha realizado correctamente, y escribimos lo siguiente:

```
C:\>RMAN CATALOG RMAN/RMAN@ORCL MSGLOG=C:\CATALOG.LOG
RMAN> CREATE CATALOG
RMAN> EXIT
```

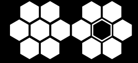
  Y para conectarse al catálogo pondremos:

```
C:\>RMAN TARGET ORCL
Recovery Manager: Release 10.1.0.2.0 - Production
Copyright (c) 1995, 2004, Oracle.  All rights reserved.
target database Password:
connected to target database: ORCL (DBID=1086216171)
RMAN> CONNECT CATALOG RMAN/RMAN@ORCL
connected to recovery catalog database
```

  Lo siguiente que se hace es registrar la base de datos destino en el catálogo, para almacenar la información sobre la base de datos. Si la base de datos no se registra en el catálogo, éste no se podrá utilizar para almacenar información sobre la base de datos:

```
RMAN> REGISTER DATABASE;
database registered in recovery catalog
starting full resync of recovery catalog
full resync complete
```

- **Software de gestión de soportes físicos.** Lo utiliza RMAN para escribir y leer en soportes físicos como cintas. Este software es proporcionado por los fabricantes de sistemas de almacenamiento y soportes físicos.



Existen una serie de comandos que nos van a permitir analizar la información del catálogo de recuperación:

- REPORT SCHEMA, visualiza la estructura de la base de datos.

- REPORT NEED BACKUP, indica de qué archivos es necesario hacer copias de seguridad:

```
RMAN> REPORT NEED BACKUP;
RMAN retention policy will be applied to the command
RMAN retention policy is set to redundancy 1
Report of files with less than 1 redundant backups

File #bkps Name
---- ----- -----------------------------------------------
1    0     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\SYSTEM01.DBF
2    0     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\UNDOTBS01.DBF
3    0     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\SYSAUX01.DBF
4    0     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF
5    0     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\EXAMPLE01.DBF
6    0     C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\EJERCIC.DBF
```

- REPORT OBSOLETE, indica qué copias de seguridad se pueden eliminar.

A. El comando BACKUP

Se puede ejecutar directamente o entre las llaves de un comando RUN. La salida puede escribirse en cinta o disco. Este comando crea juegos de copias de seguridad que normalmente contienen más de un archivo. Las copias de seguridad no contienen bloques de datos vacíos. Para extraer archivos de la copia de seguridad se realizará una operación de restauración. Los juegos de copias de seguridad de datos pueden ser incrementales o completos.

El comando puede realizar una copia de seguridad de una base de datos, un *tablespace*, un archivo de datos o de los Redo Logs archivados:

- `BACKUP DATABASE;` copia la base de datos.

- `BACKUP TABLESPACE` *nombre*; copia el *tablespace* indicado, por ejemplo: `BACKUP TABLESPACE USERS;`

- `BACKUP ARCHIVELOG ALL;` realiza copia de seguridad de los Redo Logs archivados.

- `BACKUP DATAFILE` *nombre*; realiza una copia de un archivo de datos, por ejemplo: `BACKUP DATAFILE "C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF";`

La orden `LIST BACKUP;` muestra información detallada sobre los archivos de copias de seguridad disponibles.



Caso práctico

10 Deseamos hacer un backup completo de la base de datos.

– Nos conectamos a RMAN con SYS utilizando el catálogo creado:

```
C:\> RMAN TARGET SYS/ARM@ORCL CATALOG RMAN/RMAN@ORCL
```

– Ejecutamos la orden BACKUP DATABASE, observa que copia los archivos de datos, el `control file` y el `spfile`. Observa también que lo ubica en el directorio FLASH_RECOVERY_AREA, es la ubicación predeterminada y a cada archivo de copia creado le asigna un nombre:

```
RMAN> BACKUP DATABASE;
Starting backup at 23/08/05
allocated channel: ORA_DISK_1
channel ORA_DISK_1: sid=143 devtype=DISK
channel ORA_DISK_1: starting full datafile backupset
channel ORA_DISK_1: specifying datafile(s) in backupset
input datafile fno=00001 name=C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\SYSTEM01.DBF
input datafile fno=00003 name=C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\SYSAUX01.DBF
input datafile fno=00005 name=C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\EXAMPLE01.DBF
input datafile fno=00002 name=C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\UNDOTBS01.DBF
input datafile fno=00006 name=C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\EJERCIC.DBF
input datafile fno=00004 name=C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF
channel ORA_DISK_1: starting piece 1 at 23/08/05
channel ORA_DISK_1: finished piece 1 at 23/08/05
piece handle=C:\ORACLE\PRODUCT\10.1.0\FLASH_RECOVERY_AREA\ORCL\BACKUPSET\2005_08
_23\O1_MF_NNNDF_TAG20050823T131605_1JP1CR7O_.BKP comment=NONE
channel ORA_DISK_1: backup set complete, elapsed time: 00:01:47
channel ORA_DISK_1: starting full datafile backupset
channel ORA_DISK_1: specifying datafile(s) in backupset
including current controlfile in backupset
including current SPFILE in backupset
channel ORA_DISK_1: starting piece 1 at 23/08/05
channel ORA_DISK_1: finished piece 1 at 23/08/05 piece
handle=C:\ORACLE\PRODUCT\10.1.0\FLASH_RECOVERY_AREA\ORCL\BACKUPSET\2005_08
_23\O1_MF_NCSNF_TAG20050823T131605_1JP1H3HX_.BKP comment=NONE
channel ORA_DISK_1: backup set complete, elapsed time: 00:00:06
Finished backup at 23/08/05
```

Una vez que tenemos el backup vamos a provocar un fallo y restaurar la base de datos. Utilizaremos los comandos RESTORE y RECOVER. Cerramos la base de datos y borramos dos ficheros de datos, por ejemplo, USERS01.DBF y SYSTEM01.DBF, ojo: borramos el SYSTEM.

– Desde SQL*PLUS, con el usuario SYS as SYSDBA, cerramos la base de datos SHUTDOWN IMMEDIATE. También se puede cerrar la base de datos y abrirla desde RMAN, si hemos entrado con un usuario SYSDBA.

– Borramos los archivos de datos indicados.

(Continúa)



(Continuación)

– Si intentamos conectarnos a la instancia y al catálogo nos va a dar error pues la base de datos no funciona (por eso se recomienda crear el catálogo en otra base de datos). Así pues, nos conectamos a RMAM con el usuario SYS y sin catálogo (la información de la base de datos la obtendrá del archivo de control): RMAN TARGET SYS/ARM. Montamos la base de datos, restauramos con RESTORE y recuperamos la sincronización de los archivos con RECOVER:

```
C:\>RMAN TARGET SYS/ARM
Recovery Manager: Release 10.1.0.2.0 - Production
Copyright (c) 1995, 2004, Oracle.  All rights reserved.
connected to target database (not started)
RMAN> STARTUP MOUNT;

Oracle instance started
database mounted
Total System Global Area     171966464 bytes
Fixed Size                     787988 bytes
Variable Size               145750508 bytes
Database Buffers             25165824 bytes
Redo Buffers                   262144 bytes
RMAN> RESTORE DATABASE;
Starting restore at 23/08/05
using target database controlfile instead of recovery catalog
allocated channel: ORA_DISK_1
channel ORA_DISK_1: sid=160 devtype=DISK
channel ORA_DISK_1: starting datafile backupset restore
channel ORA_DISK_1: specifying datafile(s) to restore from backup set
restoring datafile 00002 to C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\UNDOTBS01.DBF
restoring datafile 00003 to C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\SYSAUX01.DBF
restoring datafile 00004 to C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\USERS01.DBF
restoring datafile 00005 to C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\EXAMPLE01.DBF
restoring datafile 00006 to C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\EJERCIC.DBF
channel ORA_DISK_1: restored backup piece 1
piece handle=C:\ORACLE\PRODUCT\10.1.0\FLASH_RECOVERY_AREA\ORCL\BACKUPSET\2005_08
_23\O1_MF_NNNDF_TAG20050823T131605_1JP1CR7O_.BKP tag=TAG20050823T131605
channel ORA_DISK_1: restore complete
channel ORA_DISK_1: starting datafile backupset restore
channel ORA_DISK_1: specifying datafile(s) to restore from backup set
restoring datafile 00001 to C:\ORACLE\PRODUCT\10.1.0\ORADATA\ORCL\SYSTEM01.DBF
channel ORA_DISK_1: restored backup piece 1
piece handle=C:\ORACLE\PRODUCT\10.1.0\FLASH_RECOVERY_AREA\ORCL\BACKUPSET\2005_08
_23\O1_MF_NNNDF_TAG20050823T134715_1JP363WG_.BKP tag=TAG20050823T134715
channel ORA_DISK_1: restore complete
Finished restore at 23/08/05

RMAN> RECOVER DATABASE;
Starting recover at 23/08/05
using channel ORA_DISK_1
starting media recovery
archive log thread 1 sequence 260 is already on disk as file C:\ORACLE\PRODUCT\1
0.1.0\FLASH_RECOVERY_AREA\ORCL\ARCHIVELOG\2005_08_23\O1_MF_1_260_1JP3Y580_.ARC
```

(Continúa)

(Continuación)

```
archive log thread 1 sequence 261 is already on disk as file C:\ORACLE\PRODUCT\1
0.1.0\FLASH_RECOVERY_AREA\ORCL\ARCHIVELOG\2005_08_23\O1_MF_1_261_1JP9Q3QV_.ARC
archive log thread 1 sequence 262 is already on disk as file C:\ORACLE\PRODUCT\1
0.1.0\FLASH_RECOVERY_AREA\ORCL\ARCHIVELOG\2005_08_23\O1_MF_1_262_1JPB3VXM_.ARC
archive log filename=C:\ORACLE\PRODUCT\10.1.0\FLASH_RECOVERY_AREA\ORCL\ARCHIVELO
G\2005_08_23\O1_MF_1_260_1JP3Y580_.ARC thread=1 sequence=260
media recovery complete
Finished recover at 23/08/05
```

– Finalmente, la base de datos se abre llamando a SQL:

```
RMAN> SQL "ALTER DATABASE OPEN";
sql statement: alter database open
```

B. Tipos de copias de seguridad

A la hora de hacer copias de seguridad se puede elegir el tipo de copia a realizar. Existen varios tipos:

- **Las copias de seguridad completas.** Contienen todos los bloques de archivos de datos. Ademas pueden contener copias imagen, logs archivados o archivos de control.

- **Las copias de seguridad incrementales** (Nivel >=0). Contienen sólo los bloques modificados del mismo nivel, es decir, los bloques que han sido modificados desde la copia de seguridad incremental anterior, de nivel menor o igual. Una copia incremental de nivel *n* copia todos los bloques modificados desde la copia de nivel anterior o igual. La primera copia que debe crearse es de nivel 0, después se realizarán las incrementales basadas en los cambios de la copia de nivel 0. Las copias deben crearse en orden de niveles: 0, 1, 2 y 3. Oracle utiliza 4 niveles de copia incremental:

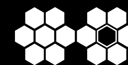
  - Nivel 0, es la copia inicial de base.

  - Nivel 1 para copias incrementales mensuales, que copia los datos cambiados desde el último mes nivel 1, o desde la última incremental base de nivel 0.

  - Nivel 2, copia semanal que incluye los bloques cambiados desde la última copia semanal nivel 2, o de nivel 1.

  - Nivel 3, es el nivel diario. Copia los bloques cambiados desde la última copia diaria nivel 3, o de nivel 2.

Las copias de seguridad incrementales de varios niveles facilitan las operaciones de recuperación puesto que durante la restauración sólo se necesita una copia de seguridad incremental de un nivel específico.





Caso práctico

11 **Estamos manteniendo una base de datos en una máquina que tiene pocos recursos y que, en hacer una copia, se tarda 4 horas. Nuestra base de datos trabaja durante las 24 horas del día, todos los días de la semana, por esa razón no se pueden realizar copias de nivel 0 (que guardan modificaciones desde el último 0) más de una vez a la semana. Se necesita que la recuperación sea rápida en caso de fallo. Así pues, se decide la siguiente estrategia de copia para la semana:**

– Realizar una copia de nivel 0 el día de la semana de menor actividad, que puede ser el domingo. A partir de esta, se van creando las incrementales.

```
BACKUP INCREMENTAL LEVEL = 0 (DATABASE);
```

– Todos los días de la semana, excepto el miércoles, se realizará una copia de nivel 2. Para guardar los cambios diarios:

```
BACKUP INCREMENTAL LEVEL = 2 (DATABASE);
```

– Se decide hacer una copia de nivel 1 el miércoles, que copiará los bloques cambiados desde el domingo, es decir, los niveles iguales o inferior, pues es un día en el que la actividad es floja. De esta forma, si hay un fallo el viernes, sólo es necesario restaurar el domingo, miércoles y jueves (los cambios del lunes y el martes van incluidos en los del miércoles).

BACKUP INCREMENTAL LEVEL = 1 (DATABASE);

Las copias de nivel 0 deben conservarse hasta que se realice otra del mismo nivel. La recuperación de copias las podemos representar como muestra la Figura 13.4.

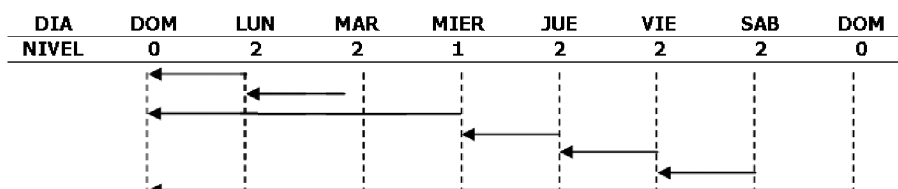


Figura 13.4. *Representación para la recuperación de copias incrementales.*

Las copias de seguridad incrementales sólo son aplicables si la base de datos de destino está abierta en modo ARCHIVELOG. También se pueden hacer copias incrementales de *tablespaces* y ficheros de datos.

– **Las copias de seguridad acumulativas incrementales** (Nivel >=0) copian todos los bloques modificados desde la copia de seguridad incremental anterior de nivel menor a *n*. Pueden tardar más que las otras copias pues escriben más bloques y generan más archivos de copia y de mayor tamaño. Este tipo de copias aumenta la velocidad de recuperación, pues el número de copias a aplicar es menor. Por ejemplo, si se realiza una copia de seguridad incremental de nivel 2, la siguiente acumulativa copiará todos los bloques modificados de nuevo más los copiados mediante la copia de seguridad incremental de nivel 2. Esto significa que sólo será necesaria una copia de seguridad incremental del mismo nivel para una recuperación completa. El comando para crear copias acumulativas es:

```
BACKUP INCREMENTAL LEVEL = 2 CUMULATIVE (DATABASE);
```

En la Figura 13.5 se muestra un esquema de las copias a realizar para recuperar una base de datos. Los martes, jueves, viernes y sábados las copias son acumulativas. Domingo, lunes y miércoles incrementales.

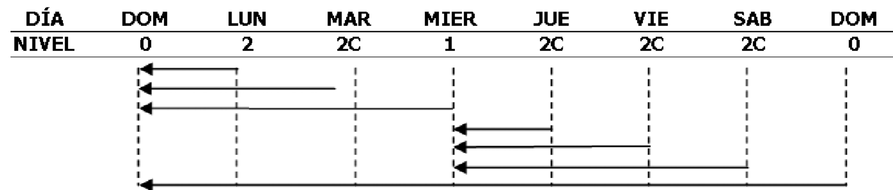


Figura 13.5. *Recuperación de copias incrementales acumulativas.*

C. Restauración y recuperación

A la hora de restaurar y recuperar hay que tener en cuenta las copias disponibles y su ubicación. Se podrán realizar restauraciones de la base de datos (vista anteriormente en el caso práctico), de *tablespaces* y de archivos de datos.

Si las copias de seguridad han sido realizadas con RMAN sólo pueden ser restauradas con RMAN. Para restaurar una base de datos en estado MOUNT utilizaremos:

- **RESTORE:** este comando restaura archivos de copias de seguridad en la ubicación predeterminada o en otra ubicación si se indica.

  `RESTORE DATABASE;` restaura la base de datos completa.

  `RESTORE TABLESPACE` *nombre*`;` restaura un *tablespace*, antes de restaurarlo hay que ponerlo offline, por ejemplo:

  ```
  RMAN> SQL "ALTER TABLESPACE USERS OFFLINE IMMEDIATE";
  RMAN> RESTORE TABLESPACE USERS;
  ```

- **RECOVER:** aplica registros de Redo Logs o copias de seguridad incrementales si las hubiera a un conjunto de copias restaurado o a una copia normal, con el fin de actualizar la base de datos a un instante específico y sincronizar todos los archivos de datos.

  `RECOVER DATABASE;` recupera y sincroniza la base de datos completa.

  `RECOVER TABLESPACE` *nombre*`;` después de hacer recover hay que ponerlo online:

  ```
  RMAN> SQL "ALTER TABLESPACE USERS OFFLINE IMMEDIATE";
  RMAN> RESTORE TABLESPACE USERS;
  RMAN> RECOVER TABLESPACE USERS;
  RMAN> SQL "ALTER TABLESPACE USERS ONLINE";
  ```

En este apartado se ha visto una pequeña parte de lo que se puede hacer con RMAN. Esta utilidad tiene una gran cantidad de comandos para realizar las copias de seguridad, restaurar y recuperar una base de datos.



13.6 Análisis de los Redo Logs

Oracle dispone de una utilidad que va a permitir analizar los archivos de Redo Logs, con lo que se podrá hacer un seguimiento de todos los cambios que se producen en la base de datos y en el diccionario de datos. La utilidad se llama **LOGMINER** y ofrece la posibilidad de procesar los archivos de Redo Log y traducir su contenido en sentencias SQL que representan las operaciones lógicas realizadas en la base de datos.

LOGMINER requiere un diccionario de datos para traducir los contenidos de los Redo Logs. El usuario SYS crea el diccionario. Para crear el diccionario de datos hacemos lo siguiente:

– Editar el archivo de inicialización de la base de datos el **SPFILEORCL.ORA** (para la instancia ORCL) que se encuentra en el directorio `C:\ORACLE\PRODUCT\10.1.0\DB_1\DATABASE`.

  Añadir la inicialización del parámetro **UTL_FILE_DIR,** donde indicaremos un directorio para ubicar el archivo del diccionario de datos, es decir, un directorio donde LOGMINER guardará las salidas de datos que genere. Crear un directorio llamado LOGMINER dentro de la instancia ORCL, y ése será el UTL_FILE_DIR: `*.utl_file_dir='C:\oracle\product\10.1.0\oradata\orcl\logminer'`. Antes de cambiar el SPFILE hacer una copia por si hubiera problemas. Observa en la Figura 13.6 cómo modificar el SPFILEORCL.

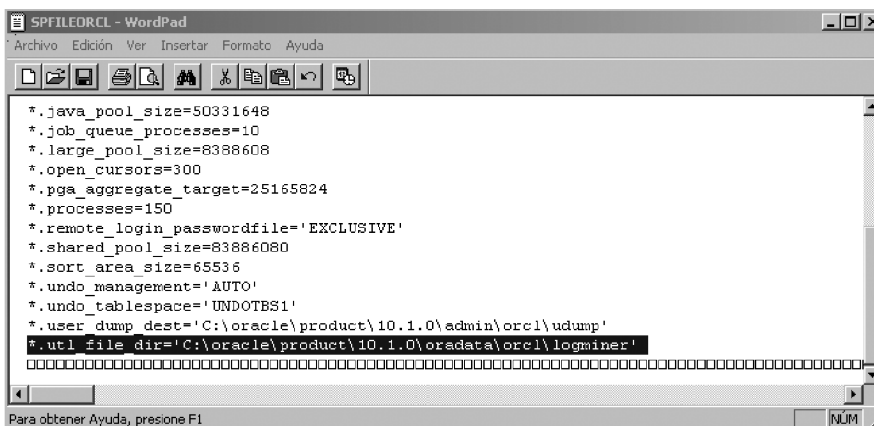


Figura 13.6. *Modificación del archivo SPFILEORCL.ORA.*

También se puede cambiar con el comando ALTER SYSTEM. En ambos casos hay que parar e iniciar la base de datos. Conectado como SYS para cambiar el parámetro, primero comprobamos el valor, ejecutamos el comando ALTER SYSTEM, paramos y arrancamos la base de datos y comprobamos si el parámetro se ha actualizado:

```
SQL> show parameter utl_
NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------------
create_stored_outlines               string
utl_file_dir                         string
```

```
SQL> alter system set utl_file_dir =
'C:\oracle\product\10.1.0\oradata\orcl\logminer' SCOPE=spfile;

System altered.
SQL> show parameter utl_
NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------------
create_stored_outlines               string
utl_file_dir                         string

SQL> shutdown immediate
Database closed.
Database dismounted.
ORACLE instance shut down.
SQL> startup
ORACLE instance started.
Total System Global Area    171966464 bytes
Fixed Size                     787988 bytes
Variable Size               145750508 bytes
Database Buffers             25165824 bytes
Redo Buffers                   262144 bytes
Database mounted.
Database opened.
SQL> show parameter utl_
NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------------
create_stored_outlines               string
utl_file_dir                         string      C:\oracle\pro-
                                                 duct\10.1.0\
                                                 oradata\orcl\
                                                 logminer
```

- Ejecutar el procedimiento DBMS_LOGMNR_D.BUILD, que se encargará de crear el archivo de diccionario que vamos a llamar ORCLDICT.ORA, y lo va a guardar en la carpeta indicada. La opción STORE_IN_FLAT_FILE significa que el archivo no va a tener formato. Una vez creado el diccionario se podrán analizar los Redo Logs. El procedimiento a ejecutar es:

```
SQL> EXECUTE
DBMS_LOGMNR_D.BUILD('ORCLDICT.ORA','C:\ORACLE\PRO-
DUCT\10.1.0\ORADATA\ORCL\LOGMINER',OPTIONS=>DBMS_LOGMNR_
D.STORE_IN_FLAT_FILE);
```

A. Sesión de LogMiner

Una vez creado el diccionario y antes de empezar a analizar los Redo Logs hay que configurar una sesión de LogMiner. Debemos especificar los archivos de log que se desean analizar. Esta utilidad puede analizar los archivos de log archivados y los online. Para ello se utilizará el procedimiento DBMS_LOGMNR.ADD_LOGFILE, que añade un archivo a



la lista. Así pues, nos conectamos con SYS y primero se crea una lista nueva con el primer log a analizar:

```
SQL> EXECUTE
DBMS_LOGMNR.ADD_LOGFILE('C:\ORACLE\PRODUCT\10.1.0\ORA-
DATA\ORCL\O1_MF_1_250.arc',DBMS_LOGMNR.NEW);
```

Si se desea añadir otro log a la lista pondremos:

```
SQL> EXECUTE
DBMS_LOGMNR.ADD_LOGFILE('C:\ORACLE\PRODUCT\10.1.0\ORA-
DATA\ORCL\REDO02.LOG',DBMS_LOGMNR.ADDFILE);
```

Y si se desea borrar un archivo de la lista utilizaremos la opción: **DBMS_LOGMNR.REMOVEFILE**

```
SQL>EXECUTE
DBMS_LOGMNR.ADD_LOGFILE('C:\ORACLE\PRODUCT\10.1.0\ORA-
DATA\ORCL\REDO 01.LOG',DBMS_LOGMNR.REMOVEFILE);
```

Una vez que se ha especificado la lista de los redo a analizar, para iniciar una sesión teclearemos:

```
SQL> EXECUTE
DBMS_LOGMNR.START_LOGMNR(DICTFILENAME=>'C:\ORACLE\PRO-
DUCT\10.1.0\ORADATA\ORCL\LOGMINER\ORCLDICT.ORA');
```

Una vez ejecutado podemos consultar las siguientes vistas:

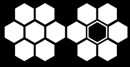
- V$LOGMNR_DICTIONARY: con la información del archivo de diccionario en uso.
- V$LOGMNR_PARAMETERS: visualiza los valores actuales de los parámetros de LogMiner.
- V$LOGMNR_LOGS: con la información de los redo a analizar, aquí se ven los que se han añadido con **DBMS_LOGMNR.ADD_LOGFILE**.
- V$LOGMNR_CONTENTS: esta vista contiene los resultados del análisis, es decir, el contenido del o de los Redo Log que se están analizando. Los datos de estas dos vistas se crean al iniciar una sesión de LogMiner.

Para cerrar una sesión teclearemos:

```
SQL> EXECUTE DBMS_LOGMNR.END_LOGMNR;
```

Al cerrar la sesión el contenido de las vistas V$LOGMNR_LOGS y V$LOGMNR_CONTENTS desaparece.

Para ver con más detalle el contenido de esta vista vamos a crear un usuario al que luego SYS le creará una tabla con los datos de esta vista. A continuación crearemos una conexión ODBC para poder abrir esta tabla en ACCESS; así pues, nos conectamos con SYS y hacemos lo siguiente:



- Creamos el usuario y le damos permisos;

```
SQL> CREATE USER USUPRUE IDENTIFIED BY USUPRUE TEMPORARY
     TABLESPACE TEMP DEFAULT TABLESPACE USERS;
SQL> ALTER USER USUPRUE QUOTA UNLIMITED ON USERS;
SQL> GRANT CONNECT, RESOURCE TO USUPRUE;
```

- Iniciamos una sesión de LogMiner, primero indicamos el log a analizar, y luego arrancamos:

```
SQL> EXECUTE
DBMS_LOGMNR.ADD_LOGFILE('C:\ORACLE\PRODUCT\10.1.0\ORA-
DATA\ORCL\RED O02.LOG',
DBMS_LOGMNR.NEW);
SQL> EXECUTE DBMS_LOGMNR.START_LOGMNR(DICTFILENAME=>'C:\ORA-
CLE\PRODUCT\10.1.0\OR ADATA\ORCL\LOGMINER\ORCLDICT.ORA');
```

- A continuación, creamos la tabla en el esquema del usuario USUPRUE:

```
SQL> CREATE TABLE USUPRUE.LOGMNR_CONTENTS AS SELECT *
FROM V$LOGMNR_CONTENTS;
```

- Cerramos Log Miner: `EXECUTE DBMS_LOGMNR.END_LOGMNR;`

- Creamos un origen de datos ODBC, para ello vamos a *Inicio/Configuración/Panel de Control/Herramientas administrativas*.

Desde aquí abrimos *Orígenes de datos* (ODBC). También se puede acceder desde el menú de Oracle 10g: *Configuration and Migration Tools / Microsoft ODBC Administrator*.

Dentro de la pestaña *DSN de Usuario*, pulsamos el botón *Agregar*, donde se muestran todos los controladores para establecer un origen de datos, hay que elegir *Oracle en OraDb10g_home1* (ver Figura 13.7).

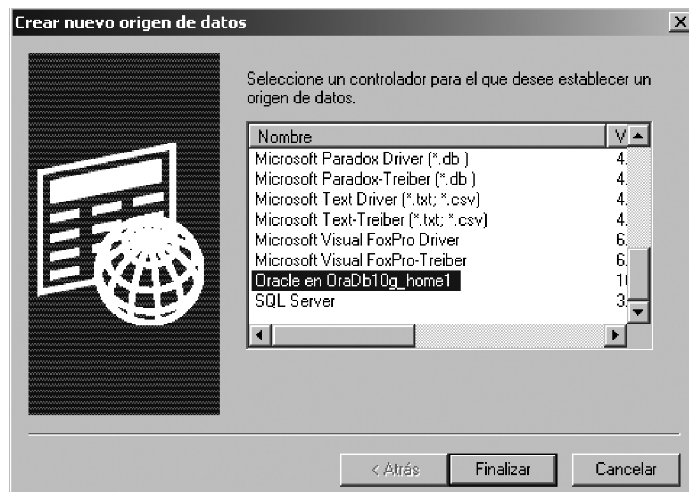
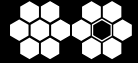


Figura 13.7. *Crear un origen de datos en Oracle 10g.*



- Seguidamente pulsamos el botón *Finalizar* y, a continuación, va a pedir la configuración de la conexión, en la que teclearemos el nombre del origen de datos y una descripción. Este nombre es el que luego veremos en la pestaña de DSN de usuario. En nombre del servicio elegimos la instancia ORCL, y en usuario ponemos USUPRUE. Ver Figura 13.8. Si se pulsa al botón *Test Connection* se realizará una prueba para ver si se ha configurado bien. Pulsamos *OK*, y el origen creado aparece en la lista de DSNs de usuario.

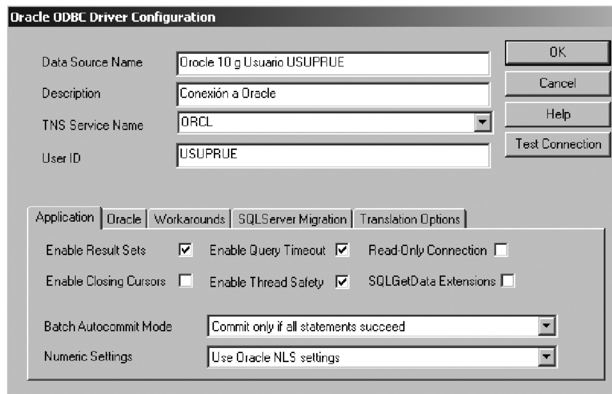


Figura 13.8. *Crear un origen de datos en Oracle 10g.*

- Una vez creado el origen de datos abrimos Access. Creamos una base de datos en blanco. Desde el menú *Archivo* elegimos *Obtener datos externos / Importar*. En la ventana de importar elegimos en tipo de archivo *Bases de datos ODBC* (ver Figura 13.9). A continuación, aparecerán todos los ODBC. Elegimos la pestaña *Origen de datos de equipo*, en ella debe figurar el nombre que pusimos en el punto anterior (ver Figura 13.10). Pulsamos *Aceptar*. Seguidamente pide la conexión a Oracle con el usuario USUPRUE y a la instancia o servicio ORCL. Una vez conectado aparecerán todos los objetos a los que tiene acceso ese usuario. Elegimos de la lista la tabla creada anteriormente USUPRUE.LOGMNR_CONTENTS y pulsamos *Aceptar* (ver figura 13.11).

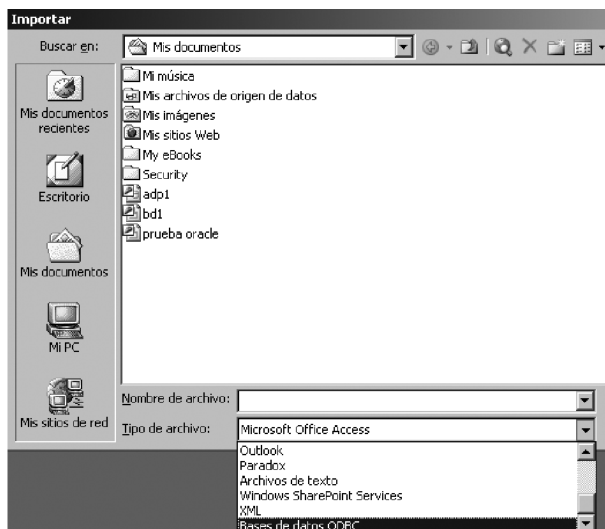


Figura 13.9. *Importar datos desde ODBC en Access.*

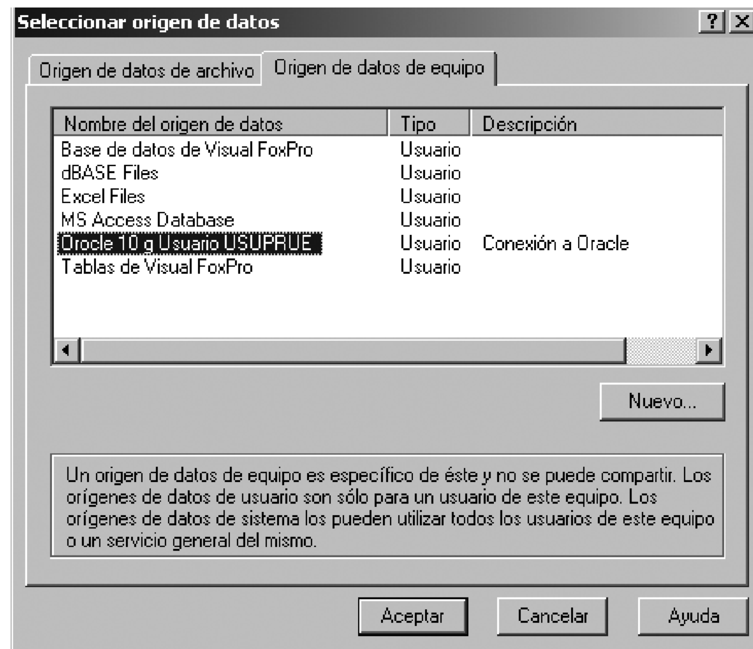


Figura 13.10. *Elección del origen asociado a la instancia ORCL.*

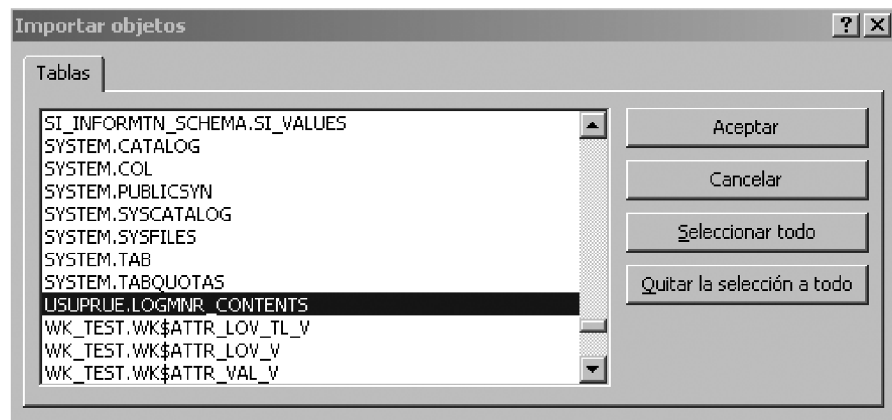


Figura 13.11. *Elección del objeto a importar.*

- Ya en Access podemos fijarnos en las columnas y los datos de esta tabla. En la Figura 13.12 se muestran operaciones del usuario SCOTT. Las columnas a destacar son:

  - SEG_OWNER: el propietario que realiza la operación.
  - TABLE_NAME y SEG_NAME: indican el nombre del objeto.
  - TABLE_SPACE: el *tablespace* donde se realiza la operación.
  - OPERATION: tipo de operación, INSERT, DELETE, una sentencia DDL.

- SQL_REDO: sentencia SQL ejecutada, aquí la columna ROLLBACK vale 1 de reconstrucción.

- SQL_UNDO: sentencia SQL de deshacer, aquí ROLLBACK vale 0.

- TIMESTAMP: Fecha de la operación realizada.

- SCN: el número de cambio del sistema.

scott : Consulta de selección

SEG_OWI	SEG_NAME	TABLE_NAME	SEG_TYPI	TABLE_SPA	SESS	USEI	ROLLE	OPERATIO	SQL_REDO
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		1	DELETE	delete from "SCOTT"."NUEEMPLE" where ROWID = 'AA
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		1	DELETE	delete from "SCOTT"."NUEEMPLE" where ROWID = 'AA
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		1	DELETE	delete from "SCOTT"."NUEEMPLE" where ROWID = 'AA
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		0	INSERT	insert into "SCOTT"."NUEEMPLE"("EMPNO","ENAME",
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		0	INSERT	insert into "SCOTT"."NUEEMPLE"("EMPNO","ENAME",
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		0	INSERT	insert into "SCOTT"."NUEEMPLE"("EMPNO","ENAME",
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		0	INSERT	insert into "SCOTT"."NUEEMPLE"("EMPNO","ENAME",
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		0	INSERT	insert into "SCOTT"."NUEEMPLE"("EMPNO","ENAME",
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		0	INSERT	insert into "SCOTT"."NUEEMPLE"("EMPNO","ENAME",
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		0	INSERT	insert into "SCOTT"."NUEEMPLE"("EMPNO","ENAME",
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		0	INSERT	insert into "SCOTT"."NUEEMPLE"("EMPNO","ENAME",
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		0	INSERT	insert into "SCOTT"."NUEEMPLE"("EMPNO","ENAME",
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		0	INSERT	insert into "SCOTT"."NUEEMPLE"("EMPNO","ENAME",
SCOTT	NUEEMPLE	NUEEMPLE	TABLE	USERS	0		0	INSERT	insert into "SCOTT"."NUEEMPLE"("EMPNO","ENAME",

Registro: 43 de 115

Figura 13.12. *Vista V$LOGMNR_CONTENTS desde Access, usuario SCOTT.*

Para acotar y filtrar valores se puede ejecutar LogMiner a partir de un SCN, o en un rango de SCNs utilizando las opciones STARTSCN y ENDSCN:

```
SQL> EXECUTE
DBMS_LOGMNR.START_LOGMNR(DICTFILENAME=>'C:\ORACLE\PRO-
DUCT\10.1.0\ORADATA\ORCL\LOGMINER\ORCLDICT.ORA',STARTSCN
=>5331831,ENDSCN=>5331839);
```

También se puede ejecutar LogMiner indicando un intervalo de tiempo utilizando las opciones STARTTIME, ENDTIME, por ejemplo, localizar todas las operaciones ocurridas el día 22/08/2005 de 10:30:00 a 11:30:00:

```
SQL> EXECUTE
DBMS_LOGMNR.START_LOGMNR(DICTFILENAME=>'C:\ORACLE\PRO-
DUCT\10.1.0\ORADATA\ORCL\LOGMINER\ORCLDICT.ORA', STARTTIME
=> TO_DATE('22/08/2005 10:30:00','DD/MM/YYYY HH:MI:SS'),
ENDTIME => TO_DATE('22/08/2005  11:30:00','DD/MM/YYYY
HH:MI:SS'));
```

13.7 Auditoría

Las auditorías se utilizan para realizar un seguimiento de todo lo que acontece en la base de datos, seguimiento de las instrucciones SQL y seguimiento de los objetos y todas sus operaciones. La base de datos admite la posibilidad de auditar todas las operaciones que tienen lugar en ella. Los registros de auditoría pueden escribirse tanto en la tabla **SYS.AUD$** como en la pista de auditoría del sistema operativo. Esta segunda opción depende del sistema operativo.

Cuando se realizan auditorías, la funcionalidad de la base de datos es dejar constancia de los comandos correctos e incorrectos. Hay que intentar restringir la auditoría, definir opciones de auditoría mínimas que satisfagan los requisitos y activar las opciones de auditoría sólo cuando sea necesario. Las auditorías generan un gran número de registros, por eso, a la hora de auditar hay que intentar reducir el número de registros generados. Así la auditoría de objetos se debe utilizar siempre que sea posible. Si hay que usar auditoría de sentencias o de privilegios hay que especificar los usuarios a auditar, auditar por sesión y no por acceso, y auditar los éxitos o los fallos, pero no ambos.

Si los registros de auditoría se guardan en la tabla SYS.AUD$, hay que pasar periódicamente sus datos a un archivo y hacer un TRUNCATE o DELETE en la tabla, pues como se almacena en el *tablespace* SYSTEM puede provocar problemas de espacio si esta tabla no se limpia. El rol **DELETE_CATALOG_ROLE** permite que un usuario borre los registros de esta tabla.

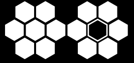
Si los registros se guardan en la pista de auditoría del sistema operativo hay que controlar y supervisar el aumento de la pista de auditoría. Si se llena las sentencias auditadas no se ejecutarán correctamente.

Para activar la auditoría en una base de datos hay que modificar el parámetro AUDIT_TRAIL, cuyos valores pueden ser los siguientes:

- NONE: valor por defecto, desactiva la auditoría.
- DB: activa la auditoría y escribe todos los registros de auditoría en la tabla SYS.AUD$, (la pista de auditoría de la base de datos).
- OS: activa la auditoría y escribe todos los registros de auditoría en la pista de auditoría del sistema operativo, si éste lo permite.

Vamos a probar la auditoría en Oracle. Para ello nos conectamos como SYS y modificamos el parámetro AUDIT_TRAIL a DB, para que tenga vigencia hay que parar y arrancar la base de datos:

```
SQL> ALTER SYSTEM SET AUDIT_TRAIL='DB' SCOPE=SPFILE;
System altered.
SQL> SHUTDOWN IMMEDIATE
SQL> STARTUP
SQL> SHOW PARAMETER AUDIT_TRAIL
NAME                                 TYPE        VALUE
------------------------------------ ----------- ---------------
audit_trail                          string      DB
```



Para ver el resultado de las auditorías se utilizarán las siguientes vistas:

- DBA_AUDIT_TRAIL: contiene todas las entradas de la pista de auditoría.

- DBA_AUDIT_OBJECT: contiene las entradas de las auditorías sobre los objetos de los esquemas y sobre privilegios del sistema.

- DBA_AUDIT_SESSION: contiene las entradas de conexión y desconexión a la base de datos.

Las vistas que contienen información acerca de las opciones de auditoría activas son las siguientes:

- ALL_DEF_AUDIT_OPTS: contiene las opciones de auditoría por defecto.

- DBA_STMT_AUDIT_OPTS: contiene las opciones de auditoría de sentencias.

- DBA_PRIV_AUDIT_OPTS: contiene las opciones de auditoría de privilegios.

- DBA_OBJ_AUDIT_OPTS: contiene las opciones de auditoría sobre objetos. Por ejemplo si se audita las inserciones y los updates en la tabla NUEEMPLE de SCOTT, las columnas INS y UPD contendrán S/S:

```
SQL> AUDIT INSERT,UPDATE ON SCOTT.NUEEMPLE;
SQL> SELECT * FROM DBA_OBJ_AUDIT_OPTS WHERE OWNER=
     'SCOTT' AND
OBJECT_NAME = 'NUEEMPLE';
```

A. Auditorías de inicio de sesión

Esta opción audita cada intento de conectarse a la base de datos. El comando para iniciar la auditoría de los inicios de sesiónes AUDIT SESSION. Para desactivar la auditoría pondremos: NOAUDIT SESSION.

AUDIT SESSION; Para todos los intentos de conexión.
AUDIT SESSION WHENEVER SUCCESSFUL; Para las conexiones correctas
AUDIT SESSION WHENEVER NOT SUCCESSFUL; Para las conexiones fallidas

La vista DBA_AUDIT_SESSION visualizará todos los intentos de conexión y de desconexión. Las columnas más significativas son:

- OS_USERNAME: usuario del sistema operativo.

- USERNAME: usuario de Oracle.

- TERMINAL: identificación del terminal utilizado.

- TIMESTAMP: fecha y hora de conexión.

- LOGOFF_TIME: fecha y hora de desconexión.

– RETURNCODE: tipo de fallo, devuelve 0 si se ha conectado, 1005 si se introduce usuario pero no contraseña, 1017 si el usuario, o la *password* es errónea. Fíjate en los errores que devuelve Oracle al conectar un usuario, en el ejemplo el usuario ficticio no existe, y Scott sí existe pero no se ha tecleado su clave:

```
SQL> CONNECT FICTICIO/A
ERROR:
ORA-01017: invalid username/password; logon denied
SQL> CONNECT SCOTT
Enter password:
ERROR:
ORA-01005: null password given; logon denied
```



Caso práctico

12 Audita todos los intentos de conexión y prueba conexiones con el usuario SCOTT, que sí existe en la base de datos y un usuario FICTICIO que no existe. Pruébalo desde SQLPLUS, después visualizaremos la vista DBA_AUDIT_SESSION. Al final cerrar la auditoría.

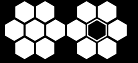
```
SQL>AUDIT SESSION;
SQL> CONNECT FICTICIO/A
SQL> CONNECT SCOTT
SQL> SELECT OS_USERNAME, USERNAME, TERMINAL, TIMESTAMP, LOGOFF_TIME,
DECODE (RETURNCODE,'0','CONECTADO', '1005','NO TECLEÓ CLAVE','1017', 'USUARIO O
CLAVE ERRÓNEA', RETURNCODE) FROM DBA_AUDIT_SESSION;
SQL> NOAUDIT SESSION;
```

B. Auditorías de sentencias y privilegios

Este tipo de auditoría contempla cualquier acción que afecte a un objeto, como puede ser: una tabla, un *tablespace*, un sinónimo, un usuario, un índice, etcétera.

Las acciones CREATE, ALTER y DROP pueden agruparse durante la auditoría de una sentencia SQL. Todos los comandos de nivel de sistema pueden auditarse, por ejemplo, para auditar todos los comandos que afectan a roles pondremos AUDIT ROLE. Esta orden auditará los comandos CREATE, ALTER, DROP y SET ROLE. Para auditar todos los comandos que afectan a usuarios pondremos AUDIT USER. Para desactivar estas opciones pondremos NOAUDIT ROLE y NOAUDIT USER.

La auditoría de privilegios audita el uso de los privilegios del sistema. Por ejemplo, si ponemos `AUDIT SELECT ANY TABLE BY SCOTT BY ACCESS;` (siempre que el usuario SCOTT utilice el privilegio SELECT ANY TABLE para consultar tablas de otros usuarios a las que no tiene concedidos privilegios) se generará un registro de auditoría. Cuando se audita se comprueban en primer lugar los privilegios de propietario, luego los de objeto y, a continuación, los de sistema.



Se utiliza la vista DBA_AUDIT_OBJECT para ver los registros de auditoría de sentencias y privilegios del sistema, y las vistas DBA_STMT_AUDIT_OPTS y DBA_PRIV_AUDIT_OPTS para ver las opciones de auditoría. A cada acción que puede auditarse se le asigna un código numérico dentro de la base de datos. A estos códigos se puede acceder a través de la vista AUDIT_ACTIONS:

```
SQL> SELECT * FROM  AUDIT_ACTIONS;
    ACTION  NAME
----------  ----------------------------
         0  UNKNOWN
         1  CREATE TABLE
         2  INSERT
         3  SELECT
         4  CREATE CLUSTER
         5  ALTER CLUSTER
         6  UPDATE
         7  DELETE
         8  DROP CLUSTER
         9  CREATE INDEX
        10  DROP INDEX
. . . . . . . . . . . . . . .
```

El formato para activar las opciones de auditoría de privilegios o sentencias es el siguiente:

```
AUDIT {sentencia | privilegio_de_sistema}[,{sentencia |
privilegio_de_sistema}]...
[BY usuario [, usuario ]...]
[BY {SESSION | ACCESS} ]
[WHENEVER [NOT] SUCCESSFUL]
```

- `BY {SESSION | ACCESS}` indica que el registro de auditoría se escribe una sola vez por sesión (éste es el valor por defecto) o cada vez que se haga un acceso a un objeto (en el caso de DDL se audita por acceso). Auditar BY ACCESS genera gran número de registros de auditoría. Esta opción se utiliza de manera limitada, por ejemplo, para controlar las acciones durante un intervalo de tiempo.

- `WHENEVER`: especifica que la auditoría se lleva a cabo cuando se completen las sentencias SQL de forma correcta o incorrecta, el valor por defecto incluye las dos opciones.

Caso práctico



13 **Nos conectamos con SYS para auditar el privilegio CREATE TABLE a SCOTT. Consultamos la vista DBA_PRIV_AUDIT_OPTS para ver si aparece esta auditoría. Borramos las filas de AUD$ para ver sólo esta auditoría:**

```
SQL> AUDIT CREATE TABLE BY SCOTT;
SQL> DELETE  FROM AUD$;
```

(Continúa)

(Continuación)

```
SQL> SELECT * FROM DBA_PRIV_AUDIT_OPTS;

USER_NAME      PROXY_NAME     PRIVILEGE        SUCCESS        FAILURE
---------      ----------     ------------     ----------     ---------
SCOTT                         CREATE TABLE     BY ACCESS      BY ACCESS
```

Luego nos conectamos con SCOTT y creamos una tabla. Seguidamente, desde SYS consultamos la vista DBA_AUDIT_OBJECT:

```
SQL> CREATE TABLE PRUEBA (V1 NUMBER(3), V2 NUMBER(3));
SQL> SELECT TERMINAL, OWNER, USERNAME, OBJ_NAME, TO_CHAR(TIMESTAMP,'DD/MON/YY
HH24:MI:SS'), RETURNCODE, ACTION_NAME FROM DBA_AUDIT_OBJECT;
```

C. Auditorías de objetos

También se pueden auditar acciones de manipulación de datos sobre objetos. Incluyen las operaciones de SELECT, INSERT, UPDATE y DELETE sobre tablas. Se utilizan las vistas DBA_AUDIT_OBJECT para ver los registros de auditoría de objetos. El formato que se utiliza es similar al anterior:

```
AUDIT sentencia [,sentencia]...
ON {[esquema.]objeto | DEFAULT}
[BY {SESSION | ACCESS} ]
[WHENEVER [NOT] SUCCESSFUL]
```



Caso práctico

14 Nos conectamos con SYS y auditamos todos los INSERT que se realicen en la tabla NUEEMPLE de SCOTT:

```
SQL> AUDIT INSERT ON SCOTT.NUEEMPLE;
```

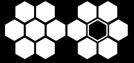
– Abrimos una sesión con SCOTT e insertamos dos registros:

```
SQL> INSERT INTO NUEEMPLE VALUES (111,'PEPITO','ENFERMERO', 7839,'09/05/80',
2000.34,100.93,10);
SQL> INSERT INTO NUEEMPLE VALUES (222,'MARÍA','MÉDICO', 7839,'10/07/83',
2500.98,170.88,20);
SQL> COMMIT;
```

– Desde SYS comprobamos lo que ha ocurrido, y lo que ha hecho SCOTT, consultamos la hora y la acción realizada, utilizamos la vista **DBA_AUDIT_OBJECT.** USERNAME es el usuario que realiza la acción, y OWNER el propietario del objeto:

```
SQL> SELECT TERMINAL, OWNER, OBJ_NAME, TO_CHAR(TIMESTAMP,'DD/MON/YY
HH24:MI:SS'), ACTION_NAME FROM DBA_AUDIT_OBJECT WHERE USERNAME='SCOTT';
```

(Continúa)



(Continuación)

Vemos que aparece un registro en la vista en la que se indica que SCOTT ha realizado un nombre de acción SESSION REC, aunque hemos insertado dos registros, sólo aparece una vez en la vista pues la auditoría se realiza por SESSION, si no se especifica BY ACCESS.

- Ahora anulamos esta auditoría y SYS audita esta misma tabla por acceso.

```
SQL> NOAUDIT INSERT ON SCOTT.NUEEMPLE;
SQL> AUDIT INSERT ON SCOTT.NUEEMPLE BY ACCESS;
```

- Abrimos de nuevo (no vale la sesión anterior) una sesión con SCOTT e insertamos dos registros:

```
SQL> INSERT INTO NUEEMPLE VALUES (333,'JUANITO','ANALISTA', 7839,'11/11/84',
2400.34,180.93,20);
SQL> INSERT INTO NUEEMPLE VALUES (444,'FELISA','TÉCNICO', 7839,'10/10/82',
2300.08,160.68,20);
SQL> COMMIT;
```

- Y desde SYS consultamos la vista DBA_AUDIT_OBJECT. Observa que ahora aparecen dos registros más y que ACTION_NAME es INSERT. Fíjate en la hora de inserción.

- Ahora SYS crea un usuario al que va a dar permiso de INSERT sobre la tabla NUEEMPLE de SCOTT. Nos conectaremos con este usuario e insertaremos dos registros. Luego consultamos la vista DBA_AUDIT_OBJECT.

```
SQL> CREATE USER PRUE_AU IDENTIFIED BY A
TEMPORARY TABLESPACE TEMP
DEFAULT TABLESPACE USERS;
SQL> ALTER USER PRUE_AU QUOTA UNLIMITED ON USERS;
SQL> GRANT CONNECT TO PRUE_AU;
SQL> GRANT INSERT ON SCOTT.NUEEMPLE TO PRUE_AU;
```

- Abrimos una sesión con PRUE_AU e insertamos dos registros, uno erróneo y el otro correcto. Por ejemplo el día de la fecha del primer registro la ponemos mal, el error es ORA-01847:

```
SQL> INSERT INTO SCOTT.NUEEMPLE VALUES (555,'ALICIA','PROFESORA', 7839,'1111/05/70'
,2000.04,140.93,20);
SQL> INSERT INTO SCOTT.NUEEMPLE VALUES (666,'JOSÉ','INGENIERO', 7839,'12/07/82'
,2600.98,170.88,30);
SQL> COMMIT;
```

- De nuevo desde SYS comprobamos lo ocurrido. En este caso, vemos los objetos propiedad de SCOTT. Fíjate que en la columna USERNAME aparecerá PRUE_AU. Observa que si se realiza un INSERT fallido, RETURNCODE devolverá el código de error, si es satisfactorio devuelve 0:

```
SQL> SELECT TERMINAL, OWNER, USERNAME, OBJ_NAME, TO_CHAR(TIMESTAMP,'DD/MON/YY
HH24:MI:SS'), RETURNCODE, ACTION_NAME FROM DBA_AUDIT_OBJECT WHERE OWNER='SCOTT';
```

Conceptos básicos



Hay dos tipos de copias de seguridad:

- **Físicas:** son copias de los datos de la base de datos. Se pueden realizar con utilidades del sistema operativo o con la utilidad RMAN de Oracle.

- **Lógicas:** son copias de distintas partes de una base de datos (*tablespaces*, tablas, esquemas, etcétera) extraídos con órdenes Oracle (EXPORT - IMPORT) y almacenados en archivos binarios. Se pueden utilizar en complemento de las copias físicas.

Las copias de seguridad de base de datos entera copian todos los archivos de datos y de control, y se pueden realizar con la base de datos cerrada o abierta. Dependiendo del momento en el que la copia de seguridad se realiza, se pueden distinguir dos tipos:

- **Copias consistentes o en frío:** realizadas cuando la base de datos está cerrada, con las opciones NORMAL o IMMEDIATE, ésta contiene las últimas modificaciones. En esta copia todas las cabeceras de los archivos de la base de datos son consistentes con el archivo de control, y cuando se restauran los archivos, la base de datos puede abrirse sin recuperación. En modo NOARCHIVELOG, sólo es válida la copia de seguridad entera consistente para la restauración y la recuperación.

- **Copias inconsistentes o en caliente:** realizadas cuando la base de datos esta en uso, o realizadas después de un cierre anormal. Este tipo se ejecuta cuando la base de datos tiene que estar disponible las 24 horas del día. En este caso las cabeceras de los archivos de datos con el de control son inconsistentes, para que la base de datos vuelva a ser consistente se necesita una recuperación. Este tipo de copia sólo se debe realizar en bases de datos en modo ARCHIVELOG. Ver Figura 13.13

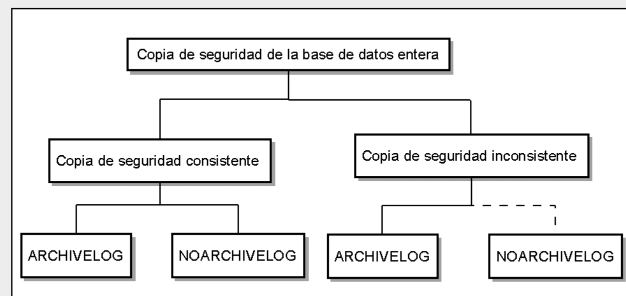
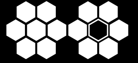


Figura 13.13. *Copia de seguridad de la base de datos entera.*



Actividades complementarias



1. **Crea un *tablespace* de nombre COMPRAS asociándole un archivo en disco llamado 'COMPRAS.ORA' de 5 Megabytes.**

2. **Modifica el *tablespace* del ejercicio 1 para que pueda auto extenderse automáticamente, sin límite de espacio en disco.**

3. **Realiza un análisis de los Redo Log en línea utilizando LogMiner.**

4. **Realiza una copia de seguridad de todos los *tablespaces*, con la base de datos abierta.**

5. **Exporta el esquema completo de un usuario de la base de datos, e importarlo en otra base de datos.**

6. **Crea una tabla y, seguidamente, crea una secuencia que genere números empezando en 10 y con incremento de 10. Inserta filas en la tabla creada utilizando la secuencia que se ha creado en alguna de las columnas.**

7. **Realiza una auditoría para controlar todos los intentos de conexión con la base de datos. Importa la tabla de auditoría AUD$ a Access para observar las operaciones realizadas.**

8. **Relaciona las vistas V$ con la información que contienen:**

   a) V$DATAFILE

   b) V$DATABASE

   c) V$LOG

   d) V$LOGFILE

   1. El Redo Log online actual.
   2. Nombre de todos los archivos de datos.
   3. Nombre de todos los Redo Logs de la base de datos.
   4. Nombre de la base de datos.

9. **Al ejecutar la orden ALTER SYSTEM SWITCH LOGFILE:**

   a) Se produce un cambio de log manual si la base de datos está en NO ARCHIVELOG.

   b) Se realiza la copia del Redo Log online si la base de datos está en ARCHIVELOG, y además se produce un cambio de log manual.

   c) Esta orden sólo se puede ejecutar si la base de datos está en modo ARCHIVELOG.

   d) Se hace una copia de seguridad de todos los Redo Log.

10. **Una copia de seguridad incremental:**

   a) Contiene los cambios producidos desde la última copia incremental del mismo nivel.

   b) Contiene los cambios producidos desde la última copia incremental del mismo o menor nivel.

   c) Contiene los cambios producidos desde la copia incremental de nivel 0.

   d) Oracle sólo permite copias completas y acumulativas.